Connolly, Thomas and Begg, Carolyn

# *Chapter 8. Advanced SQL*



ISBN: 0321523067

CHAPTER

# 8 Advanced SQL

## Chapter Objectives

In this chapter you will learn:

- How to use the SQL programming language.
- How to use SQL cursors.
- How to create stored procedures.
- How to create triggers.
- How to use triggers to enforce integrity constraints.
- The advantages and disadvantages of triggers.
- How to use recursive queries.

The previous two chapters have focused on the main language of relational DBMSs, namely SQL. In Chapter 6 we examined the data manipulation language (DML) statements of SQL: SELECT, INSERT, UPDATE, and DELETE. In Chapter 7 we examined the main data definition language (DDL) statements of SQL, such as the CREATE and ALTER TABLE statements and the CREATE VIEW statement. In this chapter we discuss some other parts of the SQL standard, namely:

- the SQL programming language (SQL/PSM);
- SQL cursors;
- stored procedures;
- triggers;
- recursive queries.

Looking ahead, in Chapter 29 we discuss the features that have been added to the SQL specification to support object-oriented data management, and in Chapter 31 we examine the features that have been added to the specification to support XML (eXtensible Markup Language), called SQL/XML:2008. Finally, in Appendix I we discuss how SQL can be embedded in high-level programming languages. The examples in this chapter use the *DreamHome* rental database shown in Figure 4.3.

## 8.1 The SQL Programming Language

The initial two versions of the SQL language had no programming constructs; that is, it was not *computationally complete*. To overcome this problem, the more recent versions of the SQL standard allow SQL to be embedded in high-level programming languages to help develop more complex database applications (see Appendix I). However, this approach produces an **impedance mismatch**, because we are mixing different programming paradigms:

- SQL is a declarative language that handles rows of data, whereas a high-level language such as C is a procedural language that can handle only one row of data at a time.
- SQL and 3GLs use different models to represent data. For example, SQL provides the built-in data types Date and Interval, which are not available in traditional programming languages. Thus, it is necessary for the application program to convert between the two representations, which is inefficient both in programming effort and in the use of runtime resources. It has been estimated that as much as 30% of programming effort and code space is expended on this type of conversion (Atkinson *et al.*, 1983). Furthermore, since we are using two different type systems, it is not possible to automatically type check the application as a whole.

It is argued that the solution to these problems is not to replace relational languages by record-level object-oriented languages, but to introduce set-level facilities into programming languages (Date, 1995). However, SQL is now a full programming language and we discuss some of the programming constructs in this section. The extensions are known as **SQL/PSM (Persistent Stored Modules)**; however, to make the discussions more concrete, we base the presentation mainly on the Oracle programming language, PL/SQL.

PL/SQL (Procedural Language/SQL) is Oracle's procedural extension to SQL. There are two versions of PL/SQL: one is part of the Oracle server, and the other is a separate engine embedded in a number of Oracle tools. They are very similar to each other and have the same programming constructs, syntax, and logic mechanisms, although PL/SQL for Oracle tools has some extensions to suit the requirements of the particular tool (for example, PL/SQL has extensions for Oracle Forms).

PL/SQL has concepts similar to modern programming languages, such as variable and constant declarations, control structures, exception handling, and modularization. PL/SQL is a block-structured language: blocks can be entirely separate or nested within one another. The basic units that constitute a PL/SQL program are procedures, functions, and anonymous *(unnamed)* blocks. As illustrated in Figure 8.1, a PL/SQL block has up to three parts:

- an optional declaration part, in which variables, constants, cursors, and exceptions are defined and possibly initialized;
- a mandatory executable part, in which the variables are manipulated;
- an optional exception part, to handle any exceptions raised during execution.

### 8.1.1 Declarations

Variables and constant variables must be declared before they can be referenced in other statements, including other declarative statements. Examples of declarations are:

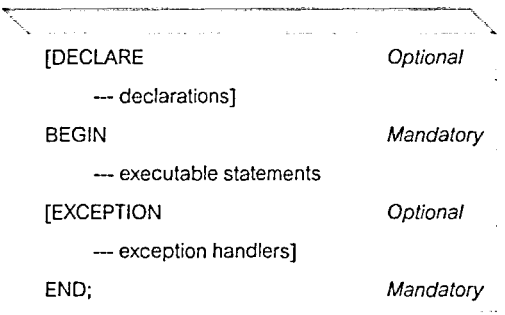


**Figure 8.1** General structure of a PL/SQL block.

```
vStaffNo VARCHAR2(5);
vRent NUMBER(6, 2) NOT NULL := 600;
MAX_PROPERTIES CONSTANT NUMBER := 100;
```

Note that it is possible to declare a variable as NOT NULL, although in this case an initial value must be assigned to the variable. It is also possible to declare a variable to be of the same type as a column in a specified table or another variable using the %TYPE attribute. For example, to declare that the vStaffNo variable is the same type as the staffNo column of the Staff table, we could write:

```
vStaffNo Staff.staffNo%TYPE;
vStaffNo1 vStaffNo%TYPE;
```

Similarly, we can declare a variable to be of the same type as an entire row of a table or view using the %ROWTYPE attribute. In this case, the fields in the record take their names and data types from the columns in the table or view. For example, to declare a vStaffRec variable to be a row from the Staff table, we could write:

```
vStaffRec Staff%ROWTYPE;
```

Note that %TYPE and %ROWTYPE are not standard SQL.

## 8.1.2 Assignments

In the executable part of a PL/SQL block, variables can be assigned in two ways: using the normal assignment statement (:=) or as the result of an SQL SELECT or FETCH statement. For example:

```
vStaffNo := 'SG14';
vRent := 500;
SELECT COUNT(*) INTO x FROM PropertyForRent WHERE staffNo = vStaffNo;
```

In the third case, the variable x is set to the result of the SELECT statement (in this case, equal to the number of properties managed by staff member SG14).

Note that in the SQL standard, an assignment uses the SET keyword at the start of the line with the "=" symbol, instead of the ":=". For example:

```
SET vStaffNo = 'SG14'
```

### 8.1.3 Control Statements

PL/SQL supports the usual conditional, iterative, and sequential flow-of-control mechanisms.

**Conditional IF statement** The IF statement has the following form:

```
IF (condition) THEN
        <SQL statement list>
[ELSIF (condition) THEN <SQL statement list>]
[ELSE <SQL statement list>]
END IF;
```

Note that the SQL standard specifies **ELSEIF** instead of **ELSIF**.

For example:

```
IF (position = 'Manager') THEN
   salary := salary*1.05;
ELSE
   salary := salary*1.03;
END IF;
```

**Conditional CASE statement** The CASE statement allows the selection of an execution path based on a set of alternatives and has the following form:

```
CASE (operand)
[WHEN (whenOperandList) | WHEN (searchCondition)
   THEN <SQL statement list>]
[ELSE <SQL statement list>]
END CASE;
```

For example:

```
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;
```

**Iteration statement (LOOP)** The LOOP statement has the following form:

```
[labelName:]
LOOP
      <SQL statement list>
      EXIT [labelName] [WHEN (condition)]
END LOOP [labelName];
```

Note that the SQL standard specifies **LEAVE** instead of **EXIT WHEN** (condition).

For example:

```
x:=1;
myLoop:
LOOP
   x := x+1;
   IF (x > 3) THEN
         EXIT myLoop;      --- exit loop immediately
END LOOP myLoop;
--- control resumes here
y := 2;
```

In this example, the loop is terminated when x becomes greater than 3 and control resumes immediately after the END LOOP keyword.

**Iteration statement (WHILE and REPEAT)** The WHILE and REPEAT statements have the following form (note that PL/SQL has no equivalent to the REPEAT loop specified in the SQL standard):

**PL/SQL**

```
WHILE (condition) LOOP
      <SQL statement list>
END LOOP [labelName];
```

**SQL**

```
WHILE (condition) DO
      <SQL statement list>
END WHILE [labelName];
REPEAT
      <SQL statement list>
UNTIL (condition)
END REPEAT [labelName];
```

**Iteration statement (FOR)** The FOR statement has the following form:

**PL/SQL**

```
FOR indexVariable
  IN lowerBound .. upperBound LOOP
  <SQL statement list>
END LOOP [labelName];
```

**SQL**

```
FOR indexVariable
  AS querySpecification DO
  <SQL statement list>
END FOR [labelName];
```

The following is an example of a FOR loop in PL/SQL:

```
DECLARE
  numberOfStaff NUMBER;
SELECT COUNT(*) INTO numberOfStaff FROM PropertyForRent
      WHERE staffNo = 'SG14'; myLoop1:
FOR iStaff IN 1 .. numberOfStaff LOOP
  .....
END LOOP
  myLoop1;
```

The following is an example of a FOR loop in standard SQL:

```
myLoop1:
FOR iStaff AS SELECT COUNT(*) FROM PropertyForRent
              WHERE staffNo = 'SG14' DO
  .....
END FOR myLoop1;
```

We present additional examples using some of these structures shortly.

## 8.1.4 Exceptions in PL/SQL

An **exception** is an identifier in PL/SQL raised during the execution of a block that terminates its main body of actions. A block always terminates when an exception is raised, although the exception handler can perform some final actions. An exception can be raised automatically by Oracle—for example, the exception NO_DATA_FOUND is raised whenever no rows are retrieved from the database in a SELECT statement. It is also possible for an exception to be raised explicitly using the RAISE statement. To handle raised exceptions, separate routines called **exception handlers** are specified.

As mentioned earlier, a user-defined exception is defined in the declarative part of a PL/SQL block. In the executable part, a check is made for the exception condition, and, if found, the exception is raised. The exception handler itself is defined at the end of the PL/SQL block. An example of exception handling is given in Figure 8.2. This example also illustrates the use of the Oracle-supplied package

```
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;
```

**Figure 8.2** Example of exception handling in PL/SQL.

DBMS_OUTPUT, which allows output from PL/SQL blocks and subprograms. The procedure put_line outputs information to a buffer in the SGA (an area of shared memory that is used to store data and control information for one Oracle instance), which can be displayed by calling the procedure get_line or by setting SERVEROUTPUT ON in SQL*Plus.

### Condition handling

The SQL Persistent Stored Modules (SQL/PSM) language includes condition handling to handle exceptions and completion conditions. Condition handling works by first defining a handler by specifying its type, the exception and completion conditions it can resolve, and the action it takes to do so (an SQL procedure statement). Condition handling also provides the ability to explicitly signal exception and completion conditions, using the SIGNAL/RESIGNAL statement.

A handler for an associated exception or completion condition can be declared using the DECLARE . . . HANDLER statement:

**DECLARE** {**CONTINUE** | **EXIT** | **UNDO**} **HANDLER**
**FOR SQLSTATE** {sqlstateValue | conditionName | **SQLEXCEPTION** |
**SQLWARNING** | **NOT FOUND**} handlerAction;

A condition name and an optional corresponding SQLSTATE value can be declared using:

**DECLARE** conditionName **CONDITION**
[**FOR SQLSTATE** sqlstateValue]

and an exception condition can be signaled or resignaled using:

**SIGNAL** sqlstateValue; or **RESIGNAL** sqlstateValue;

When a compound statement containing a handler declaration is executed, a handler is created for the associated conditions. A handler is *activated* when it is the most appropriate handler for the condition that has been raised by the SQL statement. If the handler has specified CONTINUE, then on activation it will execute the handler action before returning control to the compound statement. If the handler type is EXIT, then after executing the handler action, the handler leaves the compound statement. If the handler type is UNDO, then the handler rolls back all changes made within the compound statement, executes the associated handler action, and then returns control to the compound statement. If the handler does not complete with a *successful completion* condition, then an implicit resignal is executed, which determines whether there is another handler that can resolve the condition.

## 8.1.5 Cursors in PL/SQL

A SELECT statement can be used if the query returns *one and only one* row. To handle a query that can return an arbitrary number of rows (that is, zero, one, or more rows) PL/SQL uses **cursors** to allow the rows of a query result to be accessed one at a time. In effect, the cursor acts as a pointer to a particular row of the query result. The cursor can be advanced by 1 to access 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 cursor has been opened, the rows of the query

result can be retrieved one at a time using a FETCH statement, as opposed to a SELECT statement. (In Appendix I we see that SQL can also be embedded in high-level programming languages and that cursors are also used for handling queries that can return an arbitrary number of rows.)

**EXAMPLE 8.1 Use of cursors**

Figure 8.3 illustrates the use of a cursor to determine the properties managed by staff member SG14. In this case, the query can return an arbitrary number of rows, so a cursor must be used. The important points to note in this example are:

- In the DECLARE section, the cursor propertyCursor is defined.
- In the statements section, the cursor is first opened. Among others, this has the effect of parsing the SELECT statement specified in the CURSOR declaration, identifying the rows that satisfy the search criteria (called the *active set*), and positioning the pointer just before the first row in the active set. Note, if the query returns no rows, PL/SQL does not raise an exception when the cursor is open.
- The code then loops over each row in the active set and retrieves the current row values into output variables using the FETCH INTO statement. Each FETCH statement also advances the pointer to the next row of the active set.
- The code checks whether the cursor did not contain a row (propertyCursor%NOTFOUND) and exits the loop if no row was found (EXIT WHEN). Otherwise, it displays the property details using the DBMS_OUTPUT package and goes around the loop again.
- The cursor is closed on completion of the fetches.
- Finally, the exception block displays any error conditions encountered.

As well as %NOTFOUND, which evaluates to true if the most recent fetch does not return a row, there are some other cursor attributes that are useful:

- %FOUND: Evaluates to true if the most recent fetch returns a row (complement of %NOTFOUND).
- %ISOPEN: Evaluates to true if the cursor is open.
- %ROWCOUNT: Evaluates to the total number of rows returned so far.

**Passing parameters to cursors** PL/SQL allows cursors to be parameterized, so that the same cursor definition can be reused with different criteria. For example, we could change the cursor defined in the previous example to:

```
CURSOR propertyCursor (vStaffNo VARCHAR2) IS
        SELECT propertyNo, street, city, postcode
        FROM PropertyForRent
        WHERE staffNo = vStaffNo
        ORDER BY propertyNo;
```

and we could open the cursor using the following example statements:

```
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;
        vPostcode           PropertyForRent.postcode%TYPE;
        CURSOR propertyCursor IS
                SELECT propertyNo, street, city, postcode
                FROM PropertyForRent
                WHERE staffNo = 'SG14'
                ORDER by propertyNo;
BEGIN
-- Open the cursor to start of selection, then loop to fetch each row of the result table
    OPEN propertyCursor;
    LOOP

-- Fetch next row of the result table
        FETCH propertyCursor
                INTO vPropertyNo, vStreet, vCity, vPostcode;
        EXIT WHEN  propertyCursor%NOTFOUND;

-- Display data
        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** Using cursors in PL/SQL to process a multirow query.

**Updating rows through a cursor** It is possible to update and delete a row after it has been fetched through a cursor. In this case, to ensure that rows are not changed between declaring the cursor, opening it, and fetching the rows in the active set, the FOR UPDATE clause is added to the cursor declaration. This has the effect of locking the rows of the active set to prevent any update conflict when the cursor is opened (locking and update conflicts are discussed in Chapter 22).

For example, we may want to reassign the properties that SG14 manages to SG37. The cursor would now be declared as:

```
CURSOR propertyCursor IS
        SELECT propertyNo, street, city, postcode
        FROM PropertyForRent
        WHERE staffNo = 'SG14'
        ORDER BY propertyNo
        FOR UPDATE NOWAIT;
```

By default, if the Oracle server cannot acquire the locks on the rows in the active set in a SELECT FOR UPDATE cursor, it waits indefinitely. To prevent this, the optional NOWAIT keyword can be specified and a test can be made to see if the locking has been successful. When looping over the rows in the active set, the WHERE CURRENT OF clause is added to the SQL UPDATE or DELETE statement to indicate that the update is to be applied to the current row of the active set. For example:

```
UPDATE PropertyForRent
SET staffNo = 'SG37'
WHERE CURRENT OF propertyCursor;
  . . .
COMMIT;
```

**Cursors in the SQL standard** Cursor handling statements defined in the SQL standard are slightly different from that described previously. The interested reader is referred to Appendix I.

## 8.2 Subprograms, Stored Procedures, Functions, and Packages

**Subprograms** 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 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 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 maintainability, and they aid abstraction. A parameter 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 29.

### Packages (PL/SQL)

A **package** is a collection of procedures, functions, 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 implements 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 implementation 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 Triggers

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 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 | UPDATE [OF TriggerColumnList]
```

ON TableName
[**REFERENCING** {**OLD** | **NEW**} **AS** {OldName | NewName}
[**FOR EACH** {**ROW** | **STATEMENT**}]
[**WHEN** Condition]
<trigger action>

This is not the complete definition, but it is 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 (or possibly 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 specific 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 determines whether the action should be executed. The condition is optional but, if specified, the action will be executed only if the condition is true.
- The *action* to be taken. This block contains the SQL statements and code to be executed when a triggering statement is issued and the trigger condition evaluates to true.

There are two types 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* triggers (FOR EACH STATEMENT) that execute only once even if multiple rows are affected by the triggering event. SQL also supports INSTEAD OF triggers, which provide a transparent way of modifying views that cannot be modified directly through SQL DML statements (INSERT, UPDATE, and DELETE). These triggers are called INSTEAD OF triggers because, unlike other types of trigger, the trigger is fired *instead of* executing the original SQL statement. Triggers can also activate themselves one after the other. This can happen when the trigger action makes a change to the database that has the effect of causing another event 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 prevents 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 inserted into the PropertyForRent

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

table or an 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 referential 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 foreign 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 1 (PropertyForRent_Check_Before)***

The trigger in Figure 8.5(a) is *fired* whenever the staffNo column in the PropertyForRent table is updated. The trigger checks *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 occurring.

```
-- Before the staffNo column is updated in the PropertyForRent table, fire this trigger
-- to verify that the new foreign key value is present in the Staff table.
CREATE TRIGGER PropertyForRent_Check_Before                    <- before trigger
        BEFORE UPDATE OF staffNo ON PropertyForRent
        FOR EACH ROW WHEN (new.staffNo IS NOT NULL)            <- condition for trigger to fire
DECLARE                                                        <- row-level trigger
        dummy CHAR(5);
        invalid_staff EXCEPTION;
        valid_staff EXCEPTION;
        mutating_table EXCEPTION;
        PRAGMA EXCEPTION_INIT (mutating_table, -4091);
-- Use cursor to verify that parent key value 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 UPDATE 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;                           <- Invalid staffNo specified
        EXCEPTION
         WHEN invalid_staff THEN
                CLOSE update_cursor;
                raise_application_error(-20000, 'Invalid Staff Number' || :new.staffNo);
         WHEN valid_staff THEN
                CLOSE update_cursor;
-- A mutating table is a table that is currently being modified by an INSERT, UPDATE,
-- or DELETE statement, or 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: (a) 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 accessible to 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;                                   <-- Package to hold sequence 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 updateid NUMBER;            <-- add extra column to PropertyForRent table

-- 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 OF staffNo ON Staff                         <-- statement-level before trigger (default)
  DECLARE
        dummy NUMBER;
        BEGIN
                SELECT updatesequence.NEXTVAL               <-- set new sequence number for update
                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 UPDATE OF staffNo ON Staff                          <-- row-level after trigger
  FOR EACH ROW
  BEGIN
        UPDATE PropertyForRent SET staffNo = :new.staffNo,
                                   updateid = seqpackage.updateseq
        WHERE staffNo = :old.staffNo AND updateid IS NULL;  <-- update PropertyForRent table and set updated flag for these rows
  END;

-- Create a final statement after-trigger to reset the updateid flags
CREATE TRIGGER Cascade_StaffNo_Update3
  AFTER UPDATE OF staffNo ON Staff                          <-- statement-level after trigger; resets flags for updated rows
  BEGIN
        UPDATE PropertyForRent SET updateid = NULL
        WHERE updateid = seqpackage.updateseq;
  END;
```

**Figure 8.5(b)** Triggers for the Staff table.

***Trigger 2 (Cascade_StaffNo_Update1)***
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. Advantages of triggers include:

- *Elimination of redundant code:* Instead of placing a copy of the functionality of the trigger in every client application that requires it, the trigger is stored only once in the database.
- *Simplifying modifications:* Changing a trigger requires changing it in one place only; all the applications automatically use the updated trigger. Thus, they are only coded once, tested once, and then centrally enforced for all the applications accessing 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 benefits of security provided 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 enforced 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 transferred 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 advantages 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 significant 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 event that they are based on happens.
- *Less portable:* Although now covered by the SQL standard, most DBMSs 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 extremely 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 8.6, which stores staff numbers and the corresponding manager's staff number. Figure 8.6, which is the same as Figure 25.1(a). 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 staffNo and managerStaffNo containing all the managers 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 application may require the data to be inserted into the result table in a certain order. The recursion statement allows the specification of two orderings:

- depth-first, where each 'parent' or 'containing' item appears in the result before the items 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 example, at the end of the WITH RECURSIVE statement we could add the following clause:

```
SEARCH BREADTH FIRST BY staffNo, managerStaffNo
    SET orderColumn
```

The SET clause identifies 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 infinite loop can occur unless the cycle 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 checks 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 encountered and stops searching for further result rows. An example 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 whose element type is a row type with a column for each column in the cycle column list (staffNo and managerStaffNo in our example). Rows satisfying the query are cached in cyclePath. When a row satisfying the query is found for the first time (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 cycleMark 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 initial versions of the SQL language had no programming constructs; that is, it was not *computationally 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 assignment statements, flow of control statements (IF-THEN-ELSE-END IF; LOOP-EXIT WHEN-END LOOP; FOR-END LOOP; WHILE-END LOOP), and exceptions.
- A SELECT statement can be used if the query returns *one and only one* row. To handle a query that can return an arbitrary number of rows (that is, zero, one, or more rows), SQL uses **cursors** to allow the rows of a query result to be accessed one at a time. In effect, the cursor acts as a pointer to a particular row of the query result. The cursor can be advanced by one to access 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 cursor has been opened, the rows of the query result can be retrieved one at a time using a FETCH statement, as opposed to a SELECT statement.
- **Subprograms** are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types 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 procedure 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 *event* (or *events*) that trigger the rule, the *condition* that determines whether the action should be executed, and the *action* to be taken.
- Advantages of triggers include: eliminates redundant code, simplifies modifications, increases 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 Questions

8.1 Explain the term "impedance mismatch." Briefly describe how SQL now overcomes the impedance mismatch.

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

8.3 Describe the control statements in PL/SQL. Give examples to illustrate your answers.

8.4 Describe how the PL/SQL statements differ from the SQL standard. Give examples to illustrate 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 Discuss 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 illustrate your answers.

8.9 Discuss the advantages and disadvantages of database triggers.

## Exercises

*For the following questions, use the Hotel schema from the Exercises 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 following 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 already booked for any of the specified dates.
(d) A guest cannot make two bookings with overlapping dates.
(e) Maintain an audit table with the names and addresses of all guests who make bookings for hotels in London (do not store duplicate 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 Analyze the RDBMS that you are currently using and determine the support the system provides for SQL programming constructs, database triggers, and recursive queries. Document the differences between each system and the SQL standard.