PART IV
SQL in PL/SQL

This part of the book addresses a central element of PL/SQL code construction: the connection to the underlying Oracle database, which takes places through SQL (Structured Query Language). Chapter 14 through Chapter 16 show you how to define transactions that update, insert, merge, and delete tables in the database; query information from the database for processing in a PL/SQL program; and execute SQL statements dynamically, using native dynamic SQL (NDS).

CHAPTER 14

DML and Transaction Management

PL/SQL is tightly integrated with the Oracle database via the SQL language. From within PL/SQL, you can execute any Data Manipulation Language (DML) statements—specifically, INSERTs, UPDATEs, DELETEs, MERGEs, and, of course, queries.

You cannot, however, execute Data Definition Language (DDL) statements in PL/SQL unless you run them as dynamic SQL. This topic is covered in Chapter 16.

You can also join multiple SQL statements together logically as a *transaction*, so that they are either saved (*committed* in SQL parlance) together, or rejected in their entirety (*rolled back*). This chapter examines the SQL statements available inside PL/SQL to establish and manage transactions. It focuses on exploring the intersection point of DML and PL/SQL, answering such questions as: how can you take full advantage of DML from within the PL/SQL language? And how do you manage transactions that are created implicitly when you execute DML statements? (See "Transaction Management" on page 473.)

To appreciate the importance of transactions in Oracle, it helps to consider the ACID principle: a transaction has atomicity, consistency, isolation, and durability. These concepts are defined as follows:

Atomicity
: A transaction's changes to a state are atomic: either they all happen or none happens

Consistency
: A transaction is a correct transformation of state. The actions taken as a group do not violate any integrity constraints associated with that state.

Isolation
: Many transactions may be executing concurrently, but from any given transaction's point of view, other transactions appear to have executed before or after its own execution.

Durability
: Once a transaction completes successfully, the changes it makes to the state are made permanent and survive any subsequent failures.

You can either save a transaction by performing a COMMIT or erase it by requesting a ROLLBACK. In either case, the affected locks on resources are released (a ROLLBACK TO might release only some locks). The session can then start a new transaction. The default behavior in a PL/SQL program is that there is one transaction per session, and all changes that you make are a part of that transaction. By using a feature called *autonomous transactions*, however, you can create nested transactions within the main, session-level transaction.

DML in PL/SQL

From within any PL/SQL block of code, you can execute DML statements (INSERTs, UPDATEs, DELETEs, and MERGEs) against any and all tables and views to which you have access.

Access to these data structures is determined at the time of compilation when you're using the *definer rights model*. If you instead use the *invoker rights model* with the AUTHID CURRENT_USER compile option, access privileges are determined at runtime. See Chapter 24 for more details.

A Quick Introduction to DML

It is outside the scope of this book to provide complete reference information about the features of DML statements in the Oracle SQL language. Instead, I present a quick overview of the basic syntax, and then explore special features relating to DML inside PL/SQL, including:

- Examples of each DML statement
- Cursor attributes for DML statements
- Special PL/SQL features for DML statements, such as the RETURNING clause

For detailed information, I encourage you to peruse the Oracle documentation or a SQL-specific text.

Officially, the SELECT statement is considered a DML statement. Routinely, however, when developers refer to "DML," they almost always mean those statements that *modify* the contents of a database table. For the remainder of this chapter, DML will refer to the non-query statements of SQL.

There are four DML statements available in the SQL language:

INSERT
: Inserts one or more new rows into a table.

UPDATE
: Updates the values of one or more columns in one or more rows in a table.

DELETE
: Removes one or more rows from a table.

MERGE
: Offers nondeclarative support for an "upsert"—that is, if a row already exists for the specified column values, do an update, and otherwise, do an insert.

The INSERT statement

There are two basic types of INSERT statements:

- Insert a single row with an explicit list of values:

```
INSERT INTO table [(col1, col2, ..., coln)]
   VALUES (val1, val2, ..., valn);
```

- Insert one or more rows into a table as defined by a SELECT statement against one or more other tables:

```
INSERT INTO table [(col1, col2, ..., coln)]
   SELECT ...;
```

Let's look at some examples of INSERT statements executed within a PL/SQL block. First, I insert a new row into the book table. Notice that I do not need to specify the names of the columns if I provide a value for each column:

```
BEGIN
   INSERT INTO books
        VALUES ('1-56592-335-9',
           'Oracle PL/SQL Programming',
           'Reference for PL/SQL developers,' ||
           'including examples and best practice ' ||
           'recommendations.',
           'Feuerstein,Steven, with Bill Pribyl',
           TO_DATE ('01-SEP-1997','DD-MON-YYYY'),
```

```
          987);
END;
```

I can also list the names of the columns and provide the values as variables (including retrieval of the next available value from a sequence) instead of as literal values:

```
DECLARE
   l_isbn books.isbn%TYPE := '1-56592-335-9';
   ... other declarations of local variables ...
BEGIN
   INSERT INTO books (
        book_id, isbn, title, summary, author,
        date_published, page_count)
      VALUES (
        book_id_sequence.NEXTVAL, l_isbn, l_title, l_summary, l_author,
         l_date_published, l_page_count);
```

Native PL/SQL Support for Sequences in Oracle Database 11g

Prior to Oracle Database 11*g*, if you wanted to get the next value from a sequence, you had to execute the call to the NEXTVAL function from within a SQL statement. You could do this directly inside the INSERT statement that needed the value, as in:

```
INSERT INTO table_name VALUES (sequence_name.NEXTVAL, ...);
```

or with a SELECT from the good old dual table, as in:

```
SELECT sequence_name.NEXTVAL INTO l_primary_key FROM SYS.dual;
```

From Oracle Database 11*g* onward, however, you can now retrieve that next value (and the current value as well) with a native assignment operator. For example:

```
l_primary_key := sequence_name.NEXTVAL;
```

The UPDATE statement

With the UPDATE statement, you can update one or more columns in one or more rows. Here is the basic syntax:

```
UPDATE table
   SET col1 = val1
       [, col2 = val2, ... colN = valN]
 [WHERE where_clause];
```

The WHERE clause is optional; if you do not supply one, all rows in the table are updated. Here are some examples of UPDATEs:

- Uppercase all the titles of books in the books table:

```
UPDATE books SET title = UPPER (title);
```

- Run a utility procedure that removes the time component from the publication dates of books written by specified authors (the argument in the procedure) and uppercases the titles of those books. As you can see, you can run an UPDATE statement standalone or within a PL/SQL block:

```
PROCEDURE remove_time (author_in IN VARCHAR2)
IS
BEGIN
   UPDATE books
      SET title = UPPER (title),
          date_published = TRUNC (date_published)
    WHERE author LIKE author_in;
END;
```

The DELETE statement

You can use the DELETE statement to remove one, some, or all the rows in a table. Here is the basic syntax:

```
DELETE FROM table
 [WHERE where_clause];
```

The WHERE clause is optional in a DELETE statement. If you do not supply one, all rows in the table are deleted. Here are some examples of DELETEs:

- Delete all the books from the books table:

```
DELETE FROM books;
```

- Delete all the books from the books table that were published prior to a certain date and return the number of rows deleted:

```
PROCEDURE remove_books (
   date_in             IN       DATE,
   removal_count_out   OUT      PLS_INTEGER)
IS
BEGIN
   DELETE FROM books WHERE date_published < date_in;
   removal_count_out := SQL%ROWCOUNT;
END;
```

Of course, all these DML statements can become qualitatively more complex as you deal with real-world entities. You can, for example, update multiple columns with the contents of a subquery. And starting with Oracle9*i* Database, you can replace a table name with a *table function* that returns a result set upon which the DML statement acts (see Chapter 17).

The MERGE statement

With the MERGE statement, you specify the condition on which a match is to be evaluated, and then the two different actions to take for MATCHED and NOT MATCHED. Here is an example:

```
PROCEDURE time_use_merge (dept_in IN employees.department_id%TYPE
)
IS
BEGIN
   MERGE INTO bonuses d
        USING (SELECT employee_id, salary, department_id
                 FROM employees
                WHERE department_id = dept_in) s
           ON (d.employee_id = s.employee_id)
   WHEN MATCHED
   THEN
      UPDATE SET d.bonus = d.bonus + s.salary * .01
   WHEN NOT MATCHED
   THEN
      INSERT (d.employee_id, d.bonus)
          VALUES (s.employee_id, s.salary * 0.2
END;
```

The syntax and details of MERGE are all SQL-based, and I won't explore them further in this book. The *merge.sql* file, however, contains a more comprehensive example.

Cursor Attributes for DML Operations

Implicit cursor attributes return information about the execution of the most recent INSERT, UPDATE, DELETE, MERGE, or SELECT INTO statement. Cursor attributes for SELECT INTOs are covered in Chapter 15. In this section, I'll discuss how to take advantage of the SQL% attributes for DML statements.

First of all, remember that the values of implicit cursor attributes always refer to the most recently executed SQL statement, regardless of the block in which the implicit cursor is executed. And before Oracle opens the first SQL cursor in the session, all the implicit cursor attributes yield NULL. (The exception is %ISOPEN, which returns FALSE.)

Table 14-1 summarizes the significance of the values returned by these attributes for implicit cursors.

Table 14-1. Implicit SQL cursor attributes for DML statements

Name	Description
SQL%FOUND	Returns TRUE if one or more rows were modified (created, changed, removed) successfully
SQL%NOTFOUND	Returns TRUE if no rows were modified by the DML statement
SQL%ROWCOUNT	Returns number of rows modified by the DML statement

Name	Description
SQL%ISOPEN	Always returns FALSE for implicit cursors (and, therefore, DML statements) because the Oracle database opens and closes their cursors automatically

Now let's see how we can use cursor attributes with implicit cursors:

- Use SQL%FOUND to determine if your DML statement affected any rows. For example, from time to time an author will change his name and want a new name used for all of his books. I can create a small procedure to update the name and then report back via a Boolean variable whether any rows were modified:

```
PROCEDURE change_author_name (
   old_name_in        IN       books.author%TYPE,
   new_name_in        IN       books.author%TYPE,
   changes_made_out   OUT      BOOLEAN)
IS
BEGIN
   UPDATE books
      SET author = new_name_in
    WHERE author = old_name_in;

   changes_made_out := SQL%FOUND;
END;
```

- Use SQL%ROWCOUNT when you need to know exactly how many rows were affected by your DML statement. Here is a reworking of the preceding name-change procedure that returns a bit more information:

```
PROCEDURE change_author_name (
   old_name_in        IN       books.author%TYPE,
   new_name_in        IN       books.author%TYPE,
   rename_count_out   OUT      PLS_INTEGER)
IS
BEGIN
   UPDATE books
      SET author = new_name_in
    WHERE author = old_name_in;

   rename_count_out := SQL%ROWCOUNT;
END;
```

RETURNING Information from DML Statements

Suppose that I perform an UPDATE or DELETE, and then need to get information about the results of that statement for future processing. Rather than performing a distinct query following the DML statement, I can add a RETURNING clause to an INSERT, UPDATE, DELETE, or MERGE and retrieve that information directly into variables in my program. With the RETURNING clause, I can reduce network round-

trips, consume less server CPU time, and minimize the number of cursors opened and managed in the application.

Here are some examples that demonstrate the capabilities of this feature:

- The following very simple block shows how I use the RETURNING clause to retrieve a value (the new salary) that was computed within the UPDATE statement:

```
DECLARE
   myname  employees.last_name%TYPE;
   mysal   employees.salary%TYPE;
BEGIN
   FOR rec IN (SELECT * FROM employees)
   LOOP
      UPDATE    employees
            SET salary = salary * 1.5
          WHERE employee_id = rec.employee_id
      RETURNING salary, last_name INTO mysal, myname;

      DBMS_OUTPUT.PUT_LINE ('New salary for ' ||
         myname || ' = ' || mysal);
   END LOOP;
END;
```

- Suppose that I perform an UPDATE that modifies more than one row. In this case, I can return information not just into a single variable, but into a collection using the BULK COLLECT syntax. This technique is shown here in a FORALL statement:

```
DECLARE
   names name_varray;
   new_salaries number_varray;
BEGIN
   populate_arrays (names, new_salaries);

   FORALL indx IN names.FIRST .. names.LAST
      UPDATE compensation
         SET salary = new_salaries ( indx)
       WHERE last_name = names (indx)
       RETURNING salary BULK COLLECT INTO new_salaries;
   ...
END;
```

You'll find lots more information about the FORALL (bulk bind) statement in Chapter 21.

DML and Exception Handling

When an exception occurs in a PL/SQL block, the Oracle database does *not* roll back any of the changes made by DML statements in that block. You are the manager of the application's logical transactions, so you decide what kind of behavior should occur.

Consider the following procedure:

```
PROCEDURE empty_library (
   pre_empty_count OUT PLS_INTEGER)
IS
BEGIN

   /* tabcount implementation available in ch14_code.sql */
   pre_empty_count := tabcount ('books');

   DELETE FROM books;
   RAISE NO_DATA_FOUND;
END;
```

Notice that I set the value of the OUT parameter before I raise the exception. Now let's run an anonymous block that calls this procedure, and examine the aftereffects:

```
DECLARE
   table_count   NUMBER := -1;
BEGIN
   INSERT INTO books VALUES (...);
   empty_library (table_count);
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (tabcount ('books'));
      DBMS_OUTPUT.put_line (table_count);
END;
```

The output is:

```
0
-1
```

Notice that my rows remain deleted from the books table even though an exception was raised; the database did not perform an automatic rollback. My table_count variable, however, retains its original value.

So, it is up to you to perform rollbacks—or rather, to decide if you *want* to perform a rollback—in programs that perform DML. Here are some things to keep in mind in this regard:

- If your block is an autonomous transaction (described later in this chapter), then you *must* perform a rollback or commit (usually a rollback) when an exception is raised.
- You can use *savepoints* to control the scope of a rollback. In other words, you can roll back to a particular savepoint and thereby preserve a portion of the changes made in your session. Savepoints are also explored later in this chapter.

- If an exception propagates past the outermost block (i.e., it goes "unhandled"), then in most host execution environments for PL/SQL, like SQL*Plus, a rollback is automatically executed, reversing any outstanding changes.

DML and Records

You can use records inside INSERT and UPDATE statements. Here is an example that demonstrates the use of records in both types of statements:

```
PROCEDURE set_book_info (book_in IN books%ROWTYPE)
IS
BEGIN
   INSERT INTO books VALUES book_in;
EXCEPTION
   WHEN DUP_VAL_ON_INDEX
   THEN
      UPDATE books SET ROW = book_in
       WHERE isbn = book_in.isbn;
END;
```

This enhancement offers some compelling advantages over working with individual variables or fields within a record:

Very concise code
: You can "stay above the fray" and work completely at the record level. There is no need to declare individual variables or decompose a record into its fields when passing that data to the DML statement.

More robust code
: By working with %ROWTYPE records and not explicitly manipulating fields in those records, you make your code less likely to require maintenance as changes are made to the tables and views upon which the records are based.

In "Restrictions on record-based inserts and updates" on page 472, you will find a list of restrictions on using records in DML statements. First, let's take a look at how you can take advantage of record-based DML for the two supported statements, INSERT and UPDATE.

Record-based inserts

You can INSERT using a record with both single-row inserts and bulk inserts (via the FORALL statement). You can also use records that are based on %ROWTYPE declarations against the table to which the insert is made, or on an explicit record TYPE that is compatible with the structure of the table.

Here are some examples:

- Insert a row into the books table with a %ROWTYPE record:

```
DECLARE
   my_book books%ROWTYPE;
BEGIN
   my_book.isbn := '1-56592-335-9';
   my_book.title := 'ORACLE PL/SQL PROGRAMMING';
   my_book.summary := 'General user guide and reference';
   my_book.author := 'FEUERSTEIN, STEVEN AND BILL PRIBYL';
   my_book.page_count := 1000;

   INSERT INTO books VALUES my_book;
END;
```

Notice that you do not include parentheses around the record specifier. If you use this format:

```
INSERT INTO books VALUES (my_book); -- With parentheses, INVALID!
```

then you will get an *ORA-00947: not enough values* exception, since the program is expecting a separate expression for each column in the table.

You can also use a record based on a programmer-defined record TYPE to perform the INSERT, but that record type must be 100% compatible with the table %ROWTYPE definition. You may not, in other words, INSERT using a record that covers only a subset of the table's columns.

- Perform record-based inserts with the FORALL statement. You can also work with collections of records and insert all those records directly into a table within the FORALL statement. See Chapter 21 for more information about FORALL.

Record-based updates

You can also perform updates of an entire row using a record. The following example updates a row in the books table with a %ROWTYPE record. Notice that I use the keyword ROW to indicate that I am updating the entire row with a record:

```
/* File on web: record_updates.sql */
DECLARE
    my_book books%ROWTYPE;
 BEGIN
    my_book.isbn := '1-56592-335-9';
    my_book.title := 'ORACLE PL/SQL PROGRAMMING';
    my_book.summary := 'General user guide and reference';
    my_book.author := 'FEUERSTEIN, STEVEN AND BILL PRIBYL';
    my_book.page_count := 1000;

    UPDATE books
       SET ROW = my_book
     WHERE isbn = my_book.isbn;
 END;
```

There are some restrictions on record-based updates:

- You must update an entire row with the ROW syntax. You cannot update a subset of columns (although this may be supported in future releases). Any fields whose values are left NULL will result in a NULL value being assigned to the corresponding column.
- You cannot perform an update using a subquery.

And, in case you are wondering, you cannot create a table column called ROW.

Using records with the RETURNING clause

DML statements can include a RETURNING clause that returns column values (and expressions based on those values) from the affected row(s). You can return into a record, or even a collection of records:

```
/* File on web: record_updates.sql */
 DECLARE
    my_book_new_info books%ROWTYPE;
    my_book_return_info books%ROWTYPE;
 BEGIN
    my_book_new_info.isbn := '1-56592-335-9';
    my_book_new_info.title := 'ORACLE PL/SQL PROGRAMMING';
    my_book_new_info.summary := 'General user guide and reference';
    my_book_new_info.author := 'FEUERSTEIN, STEVEN AND BILL PRIBYL';
    my_book_new_info.page_count := 1000;

    UPDATE books
       SET ROW = my_book_new_info
     WHERE isbn = my_book_new_info.isbn
     RETURNING isbn, title, summary, author, date_published, page_count
          INTO my_book_return_info;
 END;
```

Notice that I must list each of my individual columns in the RETURNING clause. Oracle does not yet support the * syntax.

Restrictions on record-based inserts and updates

As you begin to explore these new capabilities and put them to use, keep in mind the following:

- You can use a record variable only (1) on the right side of the SET clause in UPDATEs; (2) in the VALUES clause of an INSERT; or (3) in the INTO subclause of a RETURNING clause.
- You must (and can only) use the ROW keyword on the left side of a SET clause. In this case, you may not have any other SET clauses (i.e., you may not SET a row and then SET an individual column).
- If you INSERT with a record, you may not pass individual values for columns.

- You cannot INSERT or UPDATE with a record that contains a nested record or with a function that returns a nested record.
- You cannot use records in DML statements that are executed dynamically (EXECUTE IMMEDIATE). This requires Oracle to support the binding of a PL/SQL record type into a SQL statement, and only SQL types can be bound in this way.

Transaction Management

The Oracle database provides a very robust transaction model, as you might expect from a relational database. Your application code determines what constitutes a *transaction*, which is the logical unit of work that must be either saved with a COMMIT statement or rolled back with a ROLLBACK statement. A transaction begins implicitly with the first SQL statement issued since the last COMMIT or ROLLBACK (or with the start of a session), or continues after a ROLLBACK TO SAVEPOINT.

PL/SQL provides the following statements for transaction management:

COMMIT
: Saves all outstanding changes since the last COMMIT or ROLLBACK, and releases all locks.

ROLLBACK
: Reverses the effects of all outstanding changes since the last COMMIT or ROLLBACK, and releases all locks.

ROLLBACK TO SAVEPOINT
: Reverses the effects of all changes made since the specified savepoint was established, and releases locks that were established within that range of the code.

SAVEPOINT
: Establishes a savepoint, which then allows you to perform partial ROLLBACKs.

SET TRANSACTION
: Allows you to begin a read-only or read-write session, establish an isolation level, or assign the current transaction to a specified rollback segment.

LOCK TABLE
: Allows you to lock an entire database table in the specified mode. This overrides the default row-level locking usually applied to a table.

These statements are explained in more detail in the following sections.

The COMMIT Statement

When you COMMIT, you make permanent any changes made by your session to the database in the current transaction. Once you COMMIT, your changes will be visible to other database sessions or users. The syntax for the COMMIT statement is:

```
COMMIT [WORK] [COMMENT text];
```

The WORK keyword is optional and can be used to improve readability.

The COMMENT keyword lets you specify a comment that is then associated with the current transaction. The text must be a quoted literal and can be no more than 50 characters in length. The COMMENT text is usually employed with distributed transactions, and can be handy for examining and resolving in-doubt transactions within a two-phase commit framework. It is stored in the data dictionary along with the transaction ID.

Note that COMMIT releases any row and table locks issued in your session, such as with a SELECT FOR UPDATE statement. It also erases any savepoints issued since the last COMMIT or ROLLBACK.

Once you COMMIT your changes, you cannot roll them back with a ROLLBACK statement.

The following statements are all valid uses of COMMIT:

```
COMMIT;
COMMIT WORK;
COMMIT COMMENT 'maintaining account balance'.
```

The ROLLBACK Statement

When you perform a ROLLBACK, you undo some or all changes made by your session to the database in the current transaction. Why would you want to undo changes? From an ad hoc SQL standpoint, the ROLLBACK gives you a way to erase mistakes you might have made, as in:

```
DELETE FROM orders;
```

"No, no! I meant to delete only the orders before May 2005!" No problem—just issue a ROLLBACK. From an application coding standpoint, ROLLBACK is important because it allows you to clean up or restart from a clean state when a problem occurs.

The syntax for the ROLLBACK statement is:

```
ROLLBACK [WORK] [TO [SAVEPOINT] savepoint_name];
```

There are two basic ways to use ROLLBACK: without parameters or with the TO clause to indicate a savepoint at which the ROLLBACK should stop. The parameterless ROLLBACK undoes all outstanding changes in your transaction.

The ROLLBACK TO version allows you to undo all changes and release all acquired locks that were issued after the savepoint identified by *savepoint_name*. (See the next section on the SAVEPOINT statement for more information on how to mark a savepoint in your application.)

The *savepoint_name* is an undeclared Oracle identifier. It cannot be a literal (enclosed in quotes) or variable name.

All of the following uses of ROLLBACK are valid:

```
ROLLBACK;
ROLLBACK WORK;
ROLLBACK TO begin_cleanup;
```

When you roll back to a specific savepoint, all savepoints issued after the specified *savepoint_name* are erased, but the savepoint to which you roll back is not. This means that you can restart your transaction from that point and, if necessary, roll back to that same savepoint if another error occurs.

Immediately before you execute an INSERT, UPDATE, MERGE, or DELETE, PL/SQL implicitly generates a savepoint. If your DML statement then fails, a rollback is automatically performed to that implicit savepoint. In this way, only the last DML statement is undone.

The SAVEPOINT Statement

SAVEPOINT gives a name to, and marks a point in, the processing of your transaction. This marker allows you to ROLLBACK TO that point, undoing any changes and releasing any locks issued after that savepoint, but preserving any changes and locks that occurred before you marked the savepoint.

The syntax for the SAVEPOINT statement is:

```
SAVEPOINT savepoint_name;
```

where *savepoint_name* is an undeclared identifier. This means that it must conform to the rules for an Oracle identifier (up to 30 characters in length, starting with a letter, containing letters, numbers, and #, $, or _), but that you do not need (and are not able) to declare that identifier.

Savepoints are not scoped to PL/SQL blocks. If you reuse a savepoint name within the current transaction, that savepoint is "moved" from its original position to the current point in the transaction, regardless of the procedure, function, or anonymous block in which each SAVEPOINT statement is executed. As a corollary, if you issue a savepoint inside a recursive program, a new savepoint is executed at each level of recursion, but you can only roll back to the most recently marked savepoint.

The SET TRANSACTION Statement

The SET TRANSACTION statement allows you to begin a read-only or read-write session, establish an isolation level, or assign the current transaction to a specified rollback segment. This statement must be the first SQL statement processed in a transaction, and it can appear only once. The statement comes in the following four flavors:

SET TRANSACTION READ ONLY
: This version defines the current transaction as read-only. In a read-only transaction, all subsequent queries see only those changes that were committed before the transaction began (providing a read-consistent view across tables and queries). This statement is useful when you are executing long-running multiple query reports, and you want to make sure that the data used in the reports is consistent.

SET TRANSACTION READ WRITE
: This version defines the current transaction as read-write; it is the default setting.

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE | READ COMMITTED
: This version defines how transactions that modify the database should be handled. You can specify a serializable or read-committed isolation level. When you specify SERIALIZABLE, a DML statement that attempts to modify a table that has already been modified in an uncommitted transaction will fail. To execute this command, you must set the database initialization parameter COMPATIBLE to 7.3.0 or higher.

  If you specify READ COMMITTED, a DML statement that requires row-level locks held by another transaction will wait until those row locks are released. This is the default.

SET TRANSACTION USE ROLLBACK SEGMENT `rollback_segname`
: This version assigns the current transaction to the specified rollback segment and establishes the transaction as read-write. This statement cannot be used with SET TRANSACTION READ ONLY.

Rollback segments were deprecated in favor of automatic undo management, introduced in Oracle9*i* Database.

The LOCK TABLE Statement

This statement allows you to lock an entire database table in the specified lock mode. By doing this, you can choose to deny access to that table while you perform operations against it. The syntax for this statement is:

```
LOCK TABLE table_reference_list IN lock_mode MODE [NOWAIT];
```

where *table_reference_list* is a list of one or more table references (identifying either a local table/view or a remote entity through a database link), and *lock_mode* is the mode of the lock, which can be one of the following:

- ROW SHARE
- ROW EXCLUSIVE
- SHARE UPDATE
- SHARE
- SHARE ROW EXCLUSIVE
- EXCLUSIVE

If you specify the NOWAIT keyword, the database does not wait for the lock if the table has already been locked by another user, and instead reports an error. If you leave out the NOWAIT keyword, the database waits until the table is available (and there is no set limit on how long the database will wait). Locking a table never stops other users from querying or reading the table.

The following LOCK TABLE statements show valid variations:

```
LOCK TABLE emp IN ROW EXCLUSIVE MODE;
LOCK TABLE emp, dept IN SHARE MODE NOWAIT;
LOCK TABLE scott.emp@new_york IN SHARE UPDATE MODE;
```

Whenever possible, you should rely on Oracle's default locking behavior. Use of LOCK TABLE in your application should be done as a last resort and with great care.

Autonomous Transactions

When you define a PL/SQL block as an *autonomous transaction*, you isolate the DML in that block from the caller's transaction context. That block becomes an independent transaction that is started by another transaction, referred to as the *main transaction*.

Within the autonomous transaction block, the main transaction is suspended. You perform your SQL operations, commit or roll back those operations, and resume the main transaction. This flow of transaction control is illustrated in Figure 14-1.

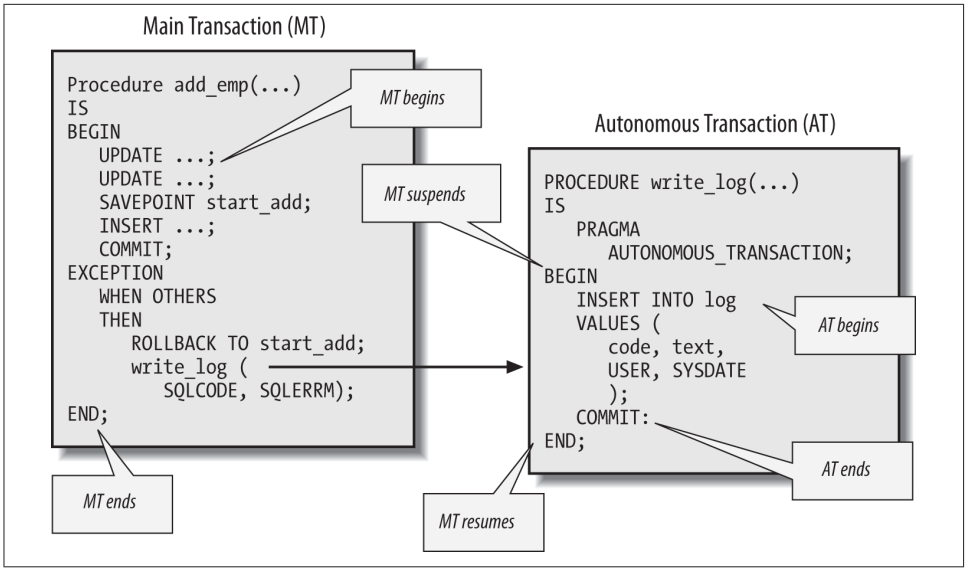

Figure 14-1. Flow of transaction control between main, nested, and autonomous transactions

Defining Autonomous Transactions

There isn't much involved in defining a PL/SQL block as an autonomous transaction. You simply include the following statement in your declaration section:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

The pragma instructs the PL/SQL compiler to establish a PL/SQL block as autonomous or independent. For the purposes of autonomous transactions, PL/SQL blocks can be any of the following:

- Top-level (but not nested) anonymous PL/SQL blocks
- Functions and procedures, defined either in a package or as standalone programs
- Methods (functions and procedures) of an object type
- Database triggers

You can put the autonomous transaction pragma anywhere in the declaration section of your PL/SQL block. You would probably be best off, however, placing it before any data structure declarations. That way, anyone reading your code will immediately identify the block as an autonomous transaction.

This pragma is the only syntax change that has been made to PL/SQL to support autonomous transactions. COMMIT, ROLLBACK, the DML statements—all the rest is as

it was before. However, these statements have a different scope of impact and visibility when executed within an autonomous transaction, and you will need to include a COMMIT or ROLLBACK in each autonomous transaction program.

Rules and Restrictions on Autonomous Transactions

While it is certainly very easy to add the autonomous transaction pragma to your code, there are some rules and restrictions on the use of this feature:

- If an autonomous transaction attempts to access a resource held by the main transaction (which has been suspended until the autonomous routine exits), a deadlock can occur in your program. Here is a simple example to demonstrate the problem. I create a procedure to perform an update, and then call it after having already updated all rows:

```
/* File on web: autondlock.sql */
PROCEDURE update_salary (dept_in IN NUMBER)
IS
   PRAGMA AUTONOMOUS_TRANSACTION;

   CURSOR myemps IS
      SELECT empno FROM emp
       WHERE deptno = dept_in
         FOR UPDATE NOWAIT;
BEGIN
   FOR rec IN myemps
   LOOP
      UPDATE emp SET sal = sal * 2
       WHERE empno = rec.empno;
   END LOOP;
   COMMIT;
END;

BEGIN
   UPDATE emp SET sal = sal * 2;
   update_salary (10);
END;
```

  The results are not pretty:

```
ERROR at line 1:
ORA-00054: resource busy and acquire with NOWAIT specified
```

- You cannot mark all the subprograms in a package (or all methods in an object type) as autonomous with a single PRAGMA declaration. You must indicate autonomous transactions explicitly in each program's declaration section in the package body. One consequence of this rule is that you cannot tell by looking at the package specification which (if any) programs will run as autonomous transactions.

- To exit without errors from an autonomous transaction program that has executed at least one INSERT, UPDATE, MERGE, or DELETE, you must perform an explicit commit or rollback. If the program (or any program called by it) has transactions pending, the runtime engine will raise the following exception and then roll back those uncommitted transactions:

  ```
  ORA-06519: active autonomous transaction detected and rolled back
  ```

- The COMMIT and ROLLBACK statements end the active autonomous transaction, but they do not force the termination of the autonomous routine. You can, in fact, have multiple COMMIT and/or ROLLBACK statements inside your autonomous block.
- You can roll back only to savepoints marked in the current transaction. When you are in an autonomous transaction, therefore, you cannot roll back to a savepoint set in the main transaction. If you try to do so, the runtime engine will raise this exception:

  ```
  ORA-01086: savepoint 'your savepoint' never established
  ```

- The TRANSACTIONS parameter in the database initialization file specifies the maximum number of transactions allowed concurrently in a session. If you use lots of autonomous transaction programs in your application, you might exceed this limit, in which case you will see the following exception:

  ```
  ORA-01574: maximum number of concurrent transactions exceeded
  ```

  In this case, increase the value for TRANSACTIONS. The default value is 75.

Transaction Visibility

The default behavior of autonomous transactions is that once a COMMIT or a ROLLBACK occurs in the autonomous transaction, those changes are visible immediately in the main transaction. But what if you want to hide those changes from the main transaction? You want them saved or undone—no question about that—but the information should not be available to the main transaction. To achieve this, use SET TRANSACTION as follows:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

The default isolation level of READ COMMITTED means that as soon as changes are committed, they are visible to the main transaction.

As is usually the case with the SET TRANSACTION statement, you must call it before you initiate your transactions (i.e., issue any SQL statements). In addition, the setting affects your entire session, not just the current program. The *autonserial.sql* script on the book's website demonstrates use of the SERIALIZABLE isolation level.

When to Use Autonomous Transactions

Where would you find autonomous transactions useful in your applications? First, let's reinforce the general principle: you will want to define your program module as an autonomous transaction whenever you want to isolate the changes made in that module from the caller's transaction context.

Here are some specific ideas:

As a logging mechanism
: On the one hand, you need to log an error to your database log table. On the other hand, you need to roll back your core transaction because of the error. And you don't want to roll back over other log entries. What's a person to do? Go autonomous! This is probably the most common motivation for PL/SQL developers to use autonomous transactions, and it's explored at the end of this section.

To perform commits and rollbacks in your database triggers
: If you define a trigger as an autonomous transaction, then you can commit and/or roll back within that trigger without affecting the transaction that fired it. Why is this valuable? You may want to take an action in the database trigger that is not affected by the ultimate disposition of the transaction that caused the trigger to fire. For example, suppose that you want to keep track of each action against a table, whether or not the action completed. You might even want to be able to detect which actions failed. See the *autontrigger*.sql* scripts on the book's website for examples of how you can apply this technique.

As reusable application components
: This usage goes to the heart of the value of autonomous transactions. As we move more and more into the dispersed, multilayered world of the Internet, it becomes ever more important to be able to offer standalone units of work (also known as *cartridges*) that get their job done without any side effects on the calling environment. Autonomous transactions play a crucial role in this area.

To avoid mutating table trigger errors for queries
: Mutating table trigger errors occur when a row-level trigger attempts to read from or write to the table from which it was fired. If, however, you make your trigger an autonomous transaction by adding the PRAGMA AUTONOMOUS_TRANSACTION statement and committing inside the body of the trigger, then you will be able to *query* the contents of the firing table—but you can see only changes already committed to the table. In other words, you will not see any changes made to the table that caused the firing of the trigger. In addition, you will still not be allowed to modify the contents of the table.

To call user-defined functions in SQL that modify tables
: Oracle lets you call your own functions inside a SQL statement, provided that this function does not update the database (and several other rules besides). If, however,

you define your function as an autonomous transaction, you will then be able to insert, update, merge, or delete inside that function as it is run from within a query. The *trcfunc.sql* script on the book's website demonstrates an application of this capability, allowing you to audit which rows of a table have been queried.

As a retry counter
: Suppose that you want to let a user try to get access to a resource *N* times before an outright rejection; you also want to keep track of attempts across connections to the database. This persistence requires a COMMIT, but one that should remain independent of the main transaction. For an example of such a utility, see *retry.pkg* and *retry.tst* on the book's website.

Building an Autonomous Logging Mechanism

A very common requirement in applications is to keep a log of errors that occur during transaction processing. The most convenient repository for this log is a database table; with a table, all the information is retained in the database, and you can use SQL to retrieve and analyze the log.

One problem with a database table log, however, is that entries in the log become a part of your transaction. If you perform a ROLLBACK (or if one is performed for you), you can easily erase your log. How frustrating! You can get fancy and use savepoints to preserve your log entries while cleaning up your transaction, but that approach is not only fancy, it is complicated. With autonomous transactions, however, logging becomes simpler, more manageable, and less error prone.

Suppose that I have a log table defined as follows:

```
/* File on web: log.pkg */
CREATE TABLE logtab (
    code INTEGER, text VARCHAR2(4000),
    created_on DATE, created_by VARCHAR2(100),
    changed_on DATE, changed_by VARCHAR2(100)
    );
```

I can use it to store errors (SQLCODE and SQLERRM) that have occurred, or even for non-error-related logging.

So I have my table. Now, how should I write to my log? Here's what you shouldn't do:

```
EXCEPTION
   WHEN OTHERS
   THEN
      DECLARE
         v_code PLS_INTEGER := SQLCODE;
         v_msg VARCHAR2(1000) := SQLERRM;
      BEGIN
         INSERT INTO logtab VALUES (
             v_code, v_msg, SYSDATE, USER, SYSDATE, USER);
```

```
        END;
END;
```

In other words, never expose your underlying logging mechanism by explicitly inserting into it your exception sections and other locations. Instead, you should build a layer of code around the table (this is known as *encapsulation*). There are three reasons to do this:

- If you ever change your table's structure, all those uses of the log table won't be disrupted.
- People can use the log table in a much easier, more consistent manner.
- You can then make that subprogram an autonomous transaction.

So here is my very simple logging package. It consists of two procedures:

```
PACKAGE log
IS
   PROCEDURE putline (code_in IN INTEGER, text_in IN VARCHAR2);
   PROCEDURE saveline (code_in IN INTEGER, text_in IN VARCHAR2);
END;
```

What is the difference between putline and saveline? The log.saveline procedure is an autonomous transaction routine; log.putline simply performs the insert. Here is the package body:

```
/* File on web: log.pkg */
PACKAGE BODY log
IS
   PROCEDURE putline (
      code_in IN INTEGER, text_in IN VARCHAR2)
   IS
   BEGIN
      INSERT INTO logtab
           VALUES (
              code_in,
              text_in,
              SYSDATE,
              USER,
              SYSDATE,
              USER
           );
   END;

   PROCEDURE saveline (
      code_in IN INTEGER, text_in IN VARCHAR2)
   IS
      PRAGMA AUTONOMOUS_TRANSACTION;
   BEGIN
      putline (code_in, text_in);
      COMMIT;
```

```
   EXCEPTION WHEN OTHERS THEN ROLLBACK;
   END;
END;
```

Here are some comments on this implementation that you might find helpful:

- The putline procedure performs the straight insert. You would probably want to add some exception handling to this program if you applied this idea in your production application.
- The saveline procedure calls the putline procedure (I don't want any redundant code), but does so from within the context of an autonomous transaction.

With this package in place, my error handler shown earlier can be as simple as this:

```
EXCEPTION
   WHEN OTHERS
   THEN
      log.saveline (SQLCODE, SQLERRM);
END;
```

No muss, no fuss. Developers don't have to concern themselves with the structure of the log table; they don't even have to know they are writing to a database table. And because I have used an autonomous transaction, they can rest assured that no matter what happens in their application, the log entry has been saved.