6

PL/SQL Static SQL

Static SQL is a PL/SQL feature that allows SQL syntax directly in a PL/SQL statement. This chapter describes static SQL and explains how to use it.

Topics

> **See Also:** "Resolution of Names in Static SQL Statements" on page B-5

Description of Static SQL

Static SQL has the same syntax as SQL, except as noted.

Topics

Statements

These are the PL/SQL static SQL statements, which have the same syntax as the corresponding SQL statements, except as noted:

- `SELECT` (this statement is also called a **query**)

  For the PL/SQL syntax, see "SELECT INTO Statement" on page 13-126.

- Data manipulation language (DML) statements:
  - `INSERT`

    For the PL/SQL syntax, see "INSERT Statement Extension" on page 13-97
  - `UPDATE`

    For the PL/SQL syntax, see "UPDATE Statement Extensions" on page 13-136

  - DELETE

    For the PL/SQL syntax, see "DELETE Statement Extension" on page 13-45

  - MERGE (for syntax, see *Oracle Database SQL Language Reference*)

    Note: *Oracle Database SQL Language Reference* defines DML differently.

- Transaction control language (TCL) statements:
  - COMMIT (for syntax, see *Oracle Database SQL Language Reference*)
  - ROLLBACK (for syntax, see *Oracle Database SQL Language Reference*)
  - SAVEPOINT (for syntax, see *Oracle Database SQL Language Reference*)
  - SET TRANSACTION (for syntax, see *Oracle Database SQL Language Reference*)
- LOCK TABLE (for syntax, see *Oracle Database SQL Language Reference*)

A PL/SQL static SQL statement can have a PL/SQL identifier wherever its SQL counterpart can have a placeholder for a bind variable. The PL/SQL identifier must identify either a variable or a formal parameter.

In Example 6–1, a PL/SQL anonymous block declares three PL/SQL variables and uses them in the static SQL statements INSERT, UPDATE, DELETE. The block also uses the static SQL statement COMMIT.

Example 6–1 Static SQL Statements

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DECLARE
  emp_id          employees_temp.employee_id%TYPE := 299;
  emp_first_name  employees_temp.first_name%TYPE  := 'Bob';
  emp_last_name   employees_temp.last_name%TYPE   := 'Henry';
BEGIN
  INSERT INTO employees_temp (employee_id, first_name, last_name)
  VALUES (emp_id, emp_first_name, emp_last_name);

  UPDATE employees_temp
  SET first_name = 'Robert'
  WHERE employee_id = emp_id;

  DELETE FROM employees_temp
  WHERE employee_id = emp_id
  RETURNING first_name, last_name
  INTO emp_first_name, emp_last_name;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE (emp_first_name || ' ' || emp_last_name);
END;
/
```

Result:

```
Robert Henry
```

To use PL/SQL identifiers for table names, column names, and so on, use the `EXECUTE IMMEDIATE` statement, explained in "Native Dynamic SQL" on page 7-2.

> **Note:** After PL/SQL code runs a DML statement, the values of some variables are undefined. For example:
>
> - After a `FETCH` or `SELECT` statement raises an exception, the values of the define variables after that statement are undefined.
> - After a DML statement that affects zero rows, the values of the `OUT` bind variables are undefined, unless the DML statement is a `BULK` or multiple-row operation.

Pseudocolumns

A pseudocolumn behaves like a table column, but it is not stored in the table. For general information about pseudocolumns, including restrictions, see *Oracle Database SQL Language Reference*.

Static SQL includes these SQL pseudocolumns:

- `CURRVAL` and `NEXTVAL`, described in "CURRVAL and NEXTVAL in PL/SQL" on page 6-3.
- `LEVEL`, described in *Oracle Database SQL Language Reference*
- `OBJECT_VALUE`, described in *Oracle Database SQL Language Reference*

  > **See Also:** "OBJECT_VALUE Pseudocolumn" on page 9-9 for information about using `OBJECT_VALUE` in triggers

- `ROWID`, described in *Oracle Database SQL Language Reference*

  > **See Also:** "Simulating CURRENT OF Clause with ROWID Pseudocolumn" on page 6-48

- `ROWNUM`, described in *Oracle Database SQL Language Reference*

CURRVAL and NEXTVAL in PL/SQL

After a sequence is created, you can access its values in SQL statements with the `CURRVAL` pseudocolumn, which returns the current value of the sequence, or the `NEXTVAL` pseudocolumn, which increments the sequence and returns the new value. (For general information about sequences, see *Oracle Database SQL Language Reference*.)

To reference these pseudocolumns, use dot notation—for example, *`sequence_name`*`.CURRVAL`. For complete syntax, see *Oracle Database SQL Language Reference*.

> **Note:** Each time you reference *`sequence_name`*`.NEXTVAL`, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

As of Oracle Database 11*g* Release 1, you can use *`sequence_name`*`.CURRVAL` and *`sequence_name`*`.NEXTVAL` in a PL/SQL expression wherever you can use a `NUMBER` expression. However:

- Using *`sequence_name`*`.CURRVAL` or *`sequence_name`*`.NEXTVAL` to provide a default value for an ADT method parameter causes a compilation error.

- PL/SQL evaluates every occurrence of *sequence_name*.CURRVAL and *sequence_name*.NEXTVAL (unlike SQL, which evaluates a sequence expression for every row in which it appears).

Example 6–2 generates a sequence number for the sequence HR.EMPLOYEES_SEQ and refers to that number in multiple statements.

Example 6–2 CURRVAL and NEXTVAL Pseudocolumns

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DROP TABLE employees_temp2;
CREATE TABLE employees_temp2 AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DECLARE
  seq_value NUMBER;
BEGIN
  -- Generate initial sequence number

  seq_value := employees_seq.NEXTVAL;

  -- Print initial sequence number:

  DBMS_OUTPUT.PUT_LINE (
    'Initial sequence value: ' || TO_CHAR(seq_value)
  );

  -- Use NEXTVAL to create unique number when inserting data:

     INSERT INTO employees_temp (employee_id, first_name, last_name)
     VALUES (employees_seq.NEXTVAL, 'Lynette', 'Smith');

  -- Use CURRVAL to store same value somewhere else:

     INSERT INTO employees_temp2 VALUES (employees_seq.CURRVAL,
                                         'Morgan', 'Smith');

  /* Because NEXTVAL values might be referenced
     by different users and applications,
     and some NEXTVAL values might not be stored in database,
     there might be gaps in sequence. */

  -- Use CURRVAL to specify record to delete:

     seq_value := employees_seq.CURRVAL;

     DELETE FROM employees_temp2
     WHERE employee_id = seq_value;

  -- Update employee_id with NEXTVAL for specified record:

     UPDATE employees_temp
     SET employee_id = employees_seq.NEXTVAL
     WHERE first_name = 'Lynette'
     AND last_name = 'Smith';
```

```
  -- Display final value of CURRVAL:

     seq_value := employees_seq.CURRVAL;

     DBMS_OUTPUT.PUT_LINE (
       'Ending sequence value: ' || TO_CHAR(seq_value)
     );
END;
/
```

Cursors

A **cursor** is a pointer to a private SQL area that stores information about processing a specific `SELECT` or DML statement.

The cursors that this chapter explains are session cursors. A **session cursor** lives in session memory until the session ends, when it ceases to exist.

A session cursor that is constructed and managed by PL/SQL is an **implicit cursor**. A session cursor that you construct and manage is an **explicit cursor**.

You can get information about any session cursor from its attributes (which you can reference in procedural statements, but not in SQL statements).

To list the session cursors that each user session currently has opened and parsed, query the dynamic performance view `V$OPEN_CURSOR`, explained in *Oracle Database Reference*.

> **Note:** Generally, PL/SQL parses an explicit cursor only the first time the session opens it and parses a SQL statement (creating an implicit cursor) only the first time the statement runs.
>
> All parsed SQL statements are cached. A SQL statement is reparsed only if it is aged out of the cache by a new SQL statement. Although you must close an explicit cursor before you can reopen it, PL/SQL need not reparse the associated query. If you close and immediately reopen an explicit cursor, PL/SQL does not reparse the associated query.

Topics

Implicit Cursors

An **implicit cursor** is a session cursor that is constructed and managed by PL/SQL. PL/SQL opens an implicit cursor every time you run a `SELECT` or DML statement. You cannot control an implicit cursor, but you can get information from its attributes.

The syntax of an implicit cursor attribute value is `SQL`*`attribute`* (therefore, an implicit cursor is also called a **SQL cursor**). `SQL`*`attribute`* always refers to the most recently run `SELECT` or DML statement. If no such statement has run, the value of `SQL`*`attribute`* is `NULL`.

An implicit cursor closes after its associated statement runs; however, its attribute values remain available until another `SELECT` or DML statement runs.

The most recently run SELECT or DML statement might be in a different scope. To save an attribute value for later use, assign it to a local variable immediately. Otherwise, other operations, such as subprogram invocations, might change the value of the attribute before you can test it.

The implicit cursor attributes are:

- SQL%ISOPEN Attribute: Is the Cursor Open?
- SQL%FOUND Attribute: Were Any Rows Affected?
- SQL%NOTFOUND Attribute: Were No Rows Affected?
- SQL%ROWCOUNT Attribute: How Many Rows Were Affected?
- SQL%BULK_ROWCOUNT (see "Getting Number of Rows Affected by FORALL Statement" on page 12-22)
- SQL%BULK_EXCEPTIONS (see "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-19)

> **See Also:** "Implicit Cursor Attribute" on page 13-92 for complete syntax and semantics

SQL%ISOPEN Attribute: Is the Cursor Open?

SQL%ISOPEN always returns FALSE, because an implicit cursor always closes after its associated statement runs.

SQL%FOUND Attribute: Were Any Rows Affected?

SQL%FOUND returns:

- NULL if no SELECT or DML statement has run
- TRUE if a SELECT statement returned one or more rows or a DML statement affected one or more rows
- FALSE otherwise

Example 6–3 uses SQL%FOUND to determine if a DELETE statement affected any rows.

Example 6–3 SQL%FOUND Implicit Cursor Attribute

```
DROP TABLE dept_temp;
CREATE TABLE dept_temp AS
  SELECT * FROM departments;

CREATE OR REPLACE PROCEDURE p (
  dept_no NUMBER
) AUTHID DEFINER AS
BEGIN
  DELETE FROM dept_temp
  WHERE department_id = dept_no;

  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE (
      'Delete succeeded for department number ' || dept_no
    );
  ELSE
    DBMS_OUTPUT.PUT_LINE ('No department number ' || dept_no);
  END IF;
END;
/
```

```
BEGIN
  p(270);
  p(400);
END;
/
```

Result:

```
Delete succeeded for department number 270
No department number 400
```

SQL%NOTFOUND Attribute: Were No Rows Affected?

SQL%NOTFOUND (the logical opposite of SQL%FOUND) returns:

- NULL if no SELECT or DML statement has run
- FALSE if a SELECT statement returned one or more rows or a DML statement affected one or more rows
- TRUE otherwise

The SQL%NOTFOUND attribute is not useful with the PL/SQL SELECT INTO statement, because:

- If the SELECT INTO statement returns no rows, PL/SQL raises the predefined exception NO_DATA_FOUND immediately, before you can check SQL%NOTFOUND.
- A SELECT INTO statement that invokes a SQL aggregate function always returns a value (possibly NULL). After such a statement, the SQL%NOTFOUND attribute is always FALSE, so checking it is unnecessary.

SQL%ROWCOUNT Attribute: How Many Rows Were Affected?

SQL%ROWCOUNT returns:

- NULL if no SELECT or DML statement has run
- Otherwise, the number of rows returned by a SELECT statement or affected by a DML statement (a PLS_INTEGER)

> **Note:** If the number of rows exceeds the maximum value for a PLS_INTEGER, then SQL%ROWCOUNT returns a negative value. For information about PLS_INTEGER, see "PLS_INTEGER and BINARY_INTEGER Data Types" on page 3-8.

Example 6–4 uses SQL%ROWCOUNT to determine the number of rows that were deleted.

Example 6–4 SQL%ROWCOUNT Implicit Cursor Attribute

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT * FROM employees;

DECLARE
  mgr_no NUMBER(6) := 122;
BEGIN
  DELETE FROM employees_temp WHERE manager_id = mgr_no;
  DBMS_OUTPUT.PUT_LINE
    ('Number of employees deleted: ' || TO_CHAR(SQL%ROWCOUNT));
END;
```

```
/
```

Result:

```
Number of employees deleted: 8
```

If a `SELECT INTO` statement without a `BULK COLLECT` clause returns multiple rows, PL/SQL raises the predefined exception `TOO_MANY_ROWS` and `SQL%ROWCOUNT` returns 1, not the actual number of rows that satisfy the query.

The value of `SQL%ROWCOUNT` attribute is unrelated to the state of a transaction. Therefore:

- When a transaction rolls back to a savepoint, the value of `SQL%ROWCOUNT` is not restored to the value it had before the savepoint.
- When an autonomous transaction ends, `SQL%ROWCOUNT` is not restored to the original value in the parent transaction.

Explicit Cursors

An **explicit cursor** is a session cursor that you construct and manage. You must declare and define an explicit cursor, giving it a name and associating it with a query (typically, the query returns multiple rows). Then you can process the query result set in either of these ways:

- Open the explicit cursor (with the `OPEN` statement), fetch rows from the result set (with the `FETCH` statement), and close the explicit cursor (with the `CLOSE` statement).
- Use the explicit cursor in a cursor `FOR LOOP` statement (see "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24).

You cannot assign a value to an explicit cursor, use it in an expression, or use it as a formal subprogram parameter or host variable. You *can* do those things with a cursor variable (see "Cursor Variables" on page 6-28).

Unlike an implicit cursor, you can reference an explicit cursor or cursor variable by its name. Therefore, an explicit cursor or cursor variable is called a **named cursor**.

Topics

Declaring and Defining Explicit Cursors

You can either declare an explicit cursor first and then define it later in the same block, subprogram, or package, or declare and define it at the same time.

An **explicit cursor declaration**, which only declares a cursor, has this syntax:

```
CURSOR cursor_name [ parameter_list ] RETURN return_type;
```

An **explicit cursor definition** has this syntax:

```
CURSOR cursor_name [ parameter_list ] [ RETURN return_type ]
  IS select_statement;
```

If you declared the cursor earlier, then the explicit cursor definition defines it; otherwise, it both declares and defines it.

Example 6–5 declares and defines three explicit cursors.

Example 6–5 Explicit Cursor Declaration and Definition

```
DECLARE
  CURSOR c1 RETURN departments%ROWTYPE;    -- Declare c1

  CURSOR c2 IS                             -- Declare and define c2
    SELECT employee_id, job_id, salary FROM employees
    WHERE salary > 2000;

  CURSOR c1 RETURN departments%ROWTYPE IS  -- Define c1,
    SELECT * FROM departments              -- repeating return type
    WHERE department_id = 110;

  CURSOR c3 RETURN locations%ROWTYPE;      -- Declare c3

  CURSOR c3 IS                             -- Define c3,
    SELECT * FROM locations                -- omitting return type
    WHERE country_id = 'JP';
BEGIN
  NULL;
END;
/
```

See Also:

- "Explicit Cursor Declaration and Definition" on page 13-57 for the complete syntax and semantics of explicit cursor declaration and definition
- "Explicit Cursors that Accept Parameters" on page 6-15

Opening and Closing Explicit Cursors

After declaring and defining an explicit cursor, you can open it with the `OPEN` statement, which does the following:

1. Allocates database resources to process the query
2. Processes the query; that is:
   1. Identifies the result set

      If the query references variables or cursor parameters, their values affect the result set. For details, see "Variables in Explicit Cursor Queries" on page 6-12 and "Explicit Cursors that Accept Parameters" on page 6-15.

   2. If the query has a `FOR UPDATE` clause, locks the rows of the result set

      For details, see "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-46.
3. Positions the cursor before the first row of the result set

You close an open explicit cursor with the CLOSE statement, thereby allowing its resources to be reused. After closing a cursor, you cannot fetch records from its result set or reference its attributes. If you try, PL/SQL raises the predefined exception INVALID_CURSOR.

You can reopen a closed cursor. You must close an explicit cursor before you try to reopen it. Otherwise, PL/SQL raises the predefined exception CURSOR_ALREADY_OPEN.

See Also:

- "OPEN Statement" on page 13-102 for its syntax and semantics
- "CLOSE Statement" on page 13-23 for its syntax and semantics

Fetching Data with Explicit Cursors

After opening an explicit cursor, you can fetch the rows of the query result set with the FETCH statement. The basic syntax of a FETCH statement that returns one row is:

```
FETCH cursor_name INTO into_clause
```

The *into_clause* is either a list of variables or a single record variable. For each column that the query returns, the variable list or record must have a corresponding type-compatible variable or field. The %TYPE and %ROWTYPE attributes are useful for declaring variables and records for use in FETCH statements.

The FETCH statement retrieves the current row of the result set, stores the column values of that row into the variables or record, and advances the cursor to the next row.

Typically, you use the FETCH statement inside a LOOP statement, which you exit when the FETCH statement runs out of rows. To detect this exit condition, use the cursor attribute %NOTFOUND (described in "%NOTFOUND Attribute: Has No Row Been Fetched?" on page 6-21). PL/SQL does not raise an exception when a FETCH statement returns no rows.

Example 6–6 fetches the result sets of two explicit cursors one row at a time, using FETCH and %NOTFOUND inside LOOP statements. The first FETCH statement retrieves column values into variables. The second FETCH statement retrieves column values into a record. The variables and record are declared with %TYPE and %ROWTYPE, respectively.

Example 6–6 FETCH Statements Inside LOOP Statements

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;

  v_lastname  employees.last_name%TYPE;  -- variable for last_name
  v_jobid     employees.job_id%TYPE;     -- variable for job_id

  CURSOR c2 IS
    SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, '[ACADFIMKSA]_M[ANGR]')
    ORDER BY job_id;

  v_employees employees%ROWTYPE;  -- record variable for row of table

BEGIN
  OPEN c1;
  LOOP  -- Fetches 2 columns into variables
```

```
    FETCH c1 INTO v_lastname, v_jobid;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE( '-----------------------------------' );

  OPEN c2;
  LOOP  -- Fetches entire row into the v_employees record
    FETCH c2 INTO v_employees;
    EXIT WHEN c2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_employees.last_name, 25, ' ') ||
                          v_employees.job_id );
  END LOOP;
  CLOSE c2;
END;
/
```

Result:

```
Atkinson                  ST_CLERK
Bell                      SH_CLERK
Bissot                    ST_CLERK
...
Walsh                     SH_CLERK
-----------------------------------
Higgins                   AC_MGR
Greenberg                 FI_MGR
Hartstein                 MK_MAN
...
Zlotkey                   SA_MAN
```

Example 6–7 fetches the first five rows of a result set into five records, using five `FETCH` statements, each of which fetches into a different record variable. The record variables are declared with `%ROWTYPE`.

Example 6–7 Fetching Same Explicit Cursor into Different Variables

```
DECLARE
  CURSOR c IS
    SELECT e.job_id, j.job_title
    FROM employees e, jobs j
    WHERE e.job_id = j.job_id AND e.manager_id = 100
    ORDER BY last_name;

  -- Record variables for rows of cursor result set:

  job1 c%ROWTYPE;
  job2 c%ROWTYPE;
  job3 c%ROWTYPE;
  job4 c%ROWTYPE;
  job5 c%ROWTYPE;

BEGIN
  OPEN c;
  FETCH c INTO job1;  -- fetches first row
  FETCH c INTO job2;  -- fetches second row
  FETCH c INTO job3;  -- fetches third row
  FETCH c INTO job4;  -- fetches fourth row
  FETCH c INTO job5;  -- fetches fifth row
  CLOSE c;
```

```
  DBMS_OUTPUT.PUT_LINE(job1.job_title || ' (' || job1.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job2.job_title || ' (' || job2.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job3.job_title || ' (' || job3.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job4.job_title || ' (' || job4.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job5.job_title || ' (' || job5.job_id || ')');
END;
/
```

Result:

```
Sales Manager (SA_MAN)
Administration Vice President (AD_VP)
Sales Manager (SA_MAN)
Stock Manager (ST_MAN)
Marketing Manager (MK_MAN)

PL/SQL procedure successfully completed.
```

See Also:

- "FETCH Statement" on page 13-71 for its complete syntax and semantics
- "FETCH Statement with BULK COLLECT Clause" on page 12-31 for information about `FETCH` statements that return more than one row at a time

Variables in Explicit Cursor Queries

An explicit cursor query can reference any variable in its scope. When you open an explicit cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

In Example 6–8, the explicit cursor query references the variable `factor`. When the cursor opens, `factor` has the value 2. Therefore, `sal_multiple` is always 2 times `sal`, despite that `factor` is incremented after every fetch.

Example 6–8 Variable in Explicit Cursor Query—No Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
  OPEN c1;  -- PL/SQL evaluates factor

  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;
```

```
  CLOSE c1;
END;
/
```

Result:

```
factor = 2
sal          = 4451
sal_multiple = 8902
factor = 3
sal          = 26460
sal_multiple = 52920
factor = 4
sal          = 18742.5
sal_multiple = 37485
factor = 5
sal          = 18742.5
sal_multiple = 37485
```

To change the result set, you must close the cursor, change the value of the variable, and then open the cursor again, as in Example 6–9.

Example 6–9 Variable in Explicit Cursor Query—Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
  OPEN c1;  -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;

  factor := factor + 1;

  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
  OPEN c1;  -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
factor = 2
sal          = 4451
sal_multiple = 8902
sal          = 26460
sal_multiple = 52920
sal          = 18742.5
sal_multiple = 37485
sal          = 18742.5
sal_multiple = 37485
factor = 3
sal          = 4451
sal_multiple = 13353
sal          = 26460
sal_multiple = 79380
sal          = 18742.5
sal_multiple = 56227.5
sal          = 18742.5
sal_multiple = 56227.5
```

When Explicit Cursor Queries Need Column Aliases

When an explicit cursor query includes a virtual column (an expression), that column must have an alias if either of the following is true:

- You use the cursor to fetch into a record that was declared with `%ROWTYPE`.
- You want to reference the virtual column in your program.

In Example 6–10, the virtual column in the explicit cursor needs an alias for both of the preceding reasons.

Example 6–10 Explicit Cursor with Virtual Column that Needs Alias

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id,
           (salary * .05) raise
    FROM employees
    WHERE job_id LIKE '%_MAN'
    ORDER BY employee_id;
  emp_rec c1%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (
      'Raise for employee #' || emp_rec.employee_id ||
      ' is $' || emp_rec.raise
    );
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
Raise for employee #114 is $550
Raise for employee #120 is $533.61
Raise for employee #121 is $520.905
Raise for employee #122 is $501.8475
Raise for employee #123 is $412.9125
```

```
Raise for employee #124 is $368.445
Raise for employee #145 is $700
Raise for employee #146 is $675
Raise for employee #147 is $600
Raise for employee #148 is $550
Raise for employee #149 is $525
Raise for employee #201 is $650
```

> **See Also:** Example 6–21, "Cursor FOR Loop References Virtual Columns"

Explicit Cursors that Accept Parameters

You can create an explicit cursor that has formal parameters, and then pass different actual parameters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere that you can use a constant. Outside the cursor query, you cannot reference formal cursor parameters.

> **Tip:** To avoid confusion, use different names for formal and actual cursor parameters.

Example 6–11 creates an explicit cursor whose two formal parameters represent a job and its maximum salary. When opened with a specified job and maximum salary, the cursor query selects the employees with that job who are overpaid (for each such employee, the query selects the first and last name and amount overpaid). Next, the example creates a procedure that prints the cursor query result set (for information about procedures, see Chapter 8, "PL/SQL Subprograms"). Finally, the example opens the cursor with one set of actual parameters, prints the result set, closes the cursor, opens the cursor with different actual parameters, prints the result set, and closes the cursor.

Example 6–11 Explicit Cursor that Accepts Parameters

```
DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_      employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('----------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Stock Clerks:');
  DBMS_OUTPUT.PUT_LINE('----------------------');
  OPEN c('ST_CLERK', 5000);
```

```
  print_overpaid;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
  DBMS_OUTPUT.PUT_LINE('------------------------------');
  OPEN c('SA_REP', 10000);
  print_overpaid;
  CLOSE c;
END;
/
```

Result:

```
----------------------
Overpaid Stock Clerks:
----------------------
Davies, Curtis (by 15.3)
Nayer, Julia (by 177.08)
Stiles, Stephen (by 177.08)
Bissot, Laura (by 338.87)
Mallin, Jason (by 338.87)
Rajs, Trenna (by 662.43)
Ladwig, Renske (by 824.21)
------------------------------
Overpaid Sales Representatives:
------------------------------
Fox, Tayler (by 80)
Tucker, Peter (by 500)
King, Janette (by 500)
Bloom, Harrison (by 500)
Vishney, Clara (by 1025)
Abel, Ellen (by 1550)
Ozer, Lisa (by 2075)

PL/SQL procedure successfully completed.
```

Topics

- Formal Cursor Parameters with Default Values
- Adding Formal Cursor Parameters with Default Values

See Also:

- "Explicit Cursor Declaration and Definition" on page 13-57 for more information about formal cursor parameters
- "OPEN Statement" on page 13-102 for more information about actual cursor parameters

Formal Cursor Parameters with Default Values When you create an explicit cursor with formal parameters, you can specify default values for them. When a formal parameter has a default value, its corresponding actual parameter is optional. If you open the cursor without specifying the actual parameter, then the formal parameter has its default value.

Example 6–12 creates an explicit cursor whose formal parameter represents a location ID. The default value of the parameter is the location ID of company headquarters.

Example 6–12 Cursor Parameters with Default Values

```
DECLARE
  CURSOR c (location NUMBER DEFAULT 1700) IS
    SELECT d.department_name,
           e.last_name manager,
           l.city
    FROM departments d, employees e, locations l
    WHERE l.location_id = location
      AND l.location_id = d.location_id
      AND d.department_id = e.department_id
    ORDER BY d.department_id;

  PROCEDURE print_depts IS
    dept_name  departments.department_name%TYPE;
    mgr_name   employees.last_name%TYPE;
    city_name  locations.city%TYPE;
  BEGIN
    LOOP
      FETCH c INTO dept_name, mgr_name, city_name;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(dept_name || ' (Manager: ' || mgr_name || ')');
    END LOOP;
  END print_depts;

BEGIN
  DBMS_OUTPUT.PUT_LINE('DEPARTMENTS AT HEADQUARTERS:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c;
  print_depts;
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('DEPARTMENTS IN CANADA:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c(1800); -- Toronto
  print_depts;
  CLOSE c;
  OPEN c(1900); -- Whitehorse
  print_depts;
  CLOSE c;
END;
/
```

Result:

```
DEPARTMENTS AT HEADQUARTERS:
--------------------------------
Administration (Manager: Whalen)
Purchasing (Manager: Colmenares)
Purchasing (Manager: Baida)
Purchasing (Manager: Himuro)
Purchasing (Manager: Raphaely)
Purchasing (Manager: Khoo)
Purchasing (Manager: Tobias)
Executive (Manager: Kochhar)
Executive (Manager: De Haan)
Executive (Manager: King)
Finance (Manager: Popp)
Finance (Manager: Greenberg)
Finance (Manager: Faviet)
```

```
Finance (Manager: Chen)
Finance (Manager: Urman)
Finance (Manager: Sciarra)
Accounting (Manager: Gietz)
Accounting (Manager: Higgins)
--------------------------------
DEPARTMENTS IN CANADA:
--------------------------------
Marketing (Manager: Hartstein)
Marketing (Manager: Fay)

PL/SQL procedure successfully completed.
```

Adding Formal Cursor Parameters with Default Values If you add formal parameters to a cursor, and you specify default values for the added parameters, then you need not change existing references to the cursor. Compare Example 6–13 to Example 6–11.

Example 6–13 Adding Formal Parameter to Existing Cursor

```
DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER, hired DATE DEFAULT '31-DEC-99') IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    AND hire_date > hired
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_      employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c('SA_REP', 10000);  -- existing reference
  print_overpaid;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('------------------------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives Hired After 2004:');
  DBMS_OUTPUT.PUT_LINE('------------------------------------------------');
  OPEN c('SA_REP', 10000, '31-DEC-04');  -- new reference
  print_overpaid;
  CLOSE c;
END;
/
```

Result:

```
------------------------------
Overpaid Sales Representatives:
------------------------------
Fox, Tayler (by 80)
Tucker, Peter (by 500)
King, Janette (by 500)
Bloom, Harrison (by 500)
Vishney, Clara (by 1025)
Abel, Ellen (by 1550)
Ozer, Lisa (by 2075)
---------------------------------------------
Overpaid Sales Representatives Hired After 2004:
---------------------------------------------
Fox, Tayler (by 80)
Tucker, Peter (by 500)
Bloom, Harrison (by 500)
Vishney, Clara (by 1025)
Ozer, Lisa (by 2075)

PL/SQL procedure successfully completed.
```

Explicit Cursor Attributes

The syntax for the value of an explicit cursor attribute is *cursor_name* immediately followed by *attribute* (for example, `c1%ISOPEN`).

> **Note:** Explicit cursors and cursor variables (named cursors) have the same attributes. This topic applies to all named cursors except where noted.

The explicit cursor attributes are:

- %ISOPEN Attribute: Is the Cursor Open?
- %FOUND Attribute: Has a Row Been Fetched?
- %NOTFOUND Attribute: Has No Row Been Fetched?
- %ROWCOUNT Attribute: How Many Rows Were Fetched?

If an explicit cursor is not open, referencing any attribute except `%ISOPEN` raises the predefined exception `INVALID_CURSOR`.

> **See Also:** "Named Cursor Attribute" on page 13-99 for complete syntax and semantics of named cursor (explicit cursor and cursor variable) attributes

%ISOPEN Attribute: Is the Cursor Open? `%ISOPEN` returns `TRUE` if its explicit cursor is open; `FALSE` otherwise.

`%ISOPEN` is useful for:

- Checking that an explicit cursor is not already open before you try to open it.

  If you try to open an explicit cursor that is already open, PL/SQL raises the predefined exception `CURSOR_ALREADY_OPEN`. You must close an explicit cursor before you can reopen it.

  > **Note:** The preceding paragraph does not apply to cursor variables.

- Checking that an explicit cursor is open before you try to close it.

Example 6–14 opens the explicit cursor `c1` only if it is not open and closes it only if it is open.

Example 6–14 %ISOPEN Explicit Cursor Attribute

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11;

  the_name employees.last_name%TYPE;
  the_salary employees.salary%TYPE;
BEGIN
  IF NOT c1%ISOPEN THEN
    OPEN c1;
  END IF;

  FETCH c1 INTO the_name, the_salary;

  IF c1%ISOPEN THEN
    CLOSE c1;
  END IF;
END;
/
```

%FOUND Attribute: Has a Row Been Fetched? `%FOUND` returns:

- `NULL` after the explicit cursor is opened but before the first fetch
- `TRUE` if the most recent fetch from the explicit cursor returned a row
- `FALSE` otherwise

`%FOUND` is useful for determining whether there is a fetched row to process.

Example 6–15 loops through a result set, printing each fetched row and exiting when there are no more rows to fetch.

Example 6–15 %FOUND Explicit Cursor Attribute

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;

  my_ename   employees.last_name%TYPE;
  my_salary  employees.salary%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_salary;
    IF c1%FOUND THEN  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE('Name = ' || my_ename || ', salary = ' || my_salary);
    ELSE  -- fetch failed
      EXIT;
    END IF;
  END LOOP;
END;
/
```

Result:

```
Name = Abel, salary = 11000
Name = Ande, salary = 6400
Name = Atkinson, salary = 3557.4
Name = Austin, salary = 4800
Name = Baer, salary = 10000
Name = Baida, salary = 2900
Name = Banda, salary = 6200
Name = Bates, salary = 7300
Name = Bell, salary = 5082
Name = Bernstein, salary = 9500
```

%NOTFOUND Attribute: Has No Row Been Fetched? `%NOTFOUND` (the logical opposite of `%FOUND`) returns:

- `NULL` after the explicit cursor is opened but before the first fetch
- `FALSE` if the most recent fetch from the explicit cursor returned a row
- `TRUE` otherwise

`%NOTFOUND` is useful for exiting a loop when `FETCH` fails to return a row, as in Example 6–16.

Example 6–16 %NOTFOUND Explicit Cursor Attribute

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;

   my_ename   employees.last_name%TYPE;
   my_salary  employees.salary%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_salary;
    IF c1%NOTFOUND THEN -- fetch failed
      EXIT;
    ELSE  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE
        ('Name = ' || my_ename || ', salary = ' || my_salary);
    END IF;
  END LOOP;
END;
/
```

Result:

```
Name = Abel, salary = 11000
Name = Ande, salary = 6400
Name = Atkinson, salary = 3557.4
Name = Austin, salary = 4800
Name = Baer, salary = 10000
Name = Baida, salary = 2900
Name = Banda, salary = 6200
Name = Bates, salary = 7300
Name = Bell, salary = 5082
Name = Bernstein, salary = 9500
```

%ROWCOUNT Attribute: How Many Rows Were Fetched? `%ROWCOUNT` returns:

- Zero after the explicit cursor is opened but before the first fetch
- Otherwise, the number of rows fetched (a `PLS_INTEGER`)

> **Note:** If the number of rows exceeds the maximum value for a `PLS_INTEGER`, then `SQL%ROWCOUNT` returns a negative value. For information about `PLS_INTEGER`, see "PLS_INTEGER and BINARY_INTEGER Data Types" on page 3-8.

Example 6–17 numbers and prints the rows that it fetches and prints a message after fetching the fifth row.

Example 6–17 %ROWCOUNT Explicit Cursor Attribute

```
DECLARE
  CURSOR c1 IS
    SELECT last_name FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;

  name  employees.last_name%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO name;
    EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
    DBMS_OUTPUT.PUT_LINE(c1%ROWCOUNT || '. ' || name);
    IF c1%ROWCOUNT = 5 THEN
       DBMS_OUTPUT.PUT_LINE('--- Fetched 5th row ---');
    END IF;
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
1. Abel
2. Ande
3. Atkinson
4. Austin
5. Baer
--- Fetched 5th row ---
6. Baida
7. Banda
8. Bates
9. Bell
10. Bernstein
```

Query Result Set Processing

In PL/SQL, as in traditional database programming, you use cursors to process query result sets. However, in PL/SQL, you can use either implicit or explicit cursors. The former need less code, but the latter are more flexible. For example, explicit cursors can accept parameters (see "Explicit Cursors that Accept Parameters" on page 6-15).

The following PL/SQL statements use implicit cursors that PL/SQL defines and manages for you:

- `SELECT INTO`
- Implicit cursor `FOR LOOP`

The following PL/SQL statements use explicit cursors:

- Explicit cursor `FOR LOOP`

  You define the explicit cursor, but PL/SQL manages it while the statement runs.
- `OPEN`, `FETCH`, and `CLOSE`

  You define and manage the explicit cursor.

Topics

- Query Result Set Processing With SELECT INTO Statements
- Query Result Set Processing With Cursor FOR LOOP Statements
- Query Result Set Processing With Explicit Cursors, OPEN, FETCH, and CLOSE
- Query Result Set Processing with Subqueries

> **Note:** If a query returns no rows, PL/SQL raises the exception `NO_DATA_FOUND`. For information about handling exceptions, see "Exception Handler" on page 13-50.

Query Result Set Processing With SELECT INTO Statements

Using an implicit cursor, the `SELECT INTO` statement retrieves values from one or more database tables (as the SQL `SELECT` statement does) and stores them in variables (which the SQL `SELECT` statement does not do).

Topics

- Single-Row Result Sets
- Large Multiple-Row Result Sets

> **See Also:** "SELECT INTO Statement" on page 13-126 for its complete syntax and semantics

Single-Row Result Sets

If you expect the query to return only one row, then use the `SELECT INTO` statement to store values from that row in either one or more scalar variables (see "Assigning Values to Variables with the SELECT INTO Statement" on page 2-22) or one record variable (see "SELECT INTO Statement for Assigning Row to Record Variable" on page 5-50).

If the query might return multiple rows, but you care about only the *n*th row, then restrict the result set to that row with the clause `WHERE ROWNUM=`*n*. For more information about the `ROWNUM` pseudocolumn, see *Oracle Database SQL Language Reference*.

Large Multiple-Row Result Sets

If you must assign a large quantity of table data to variables, Oracle recommends using the `SELECT INTO` statement with the `BULK COLLECT` clause. This statement retrieves an entire result set into one or more collection variables. For more

information, see "SELECT INTO Statement with BULK COLLECT Clause" on page 12-24.

Query Result Set Processing With Cursor FOR LOOP Statements

The cursor `FOR LOOP` statement lets you run a `SELECT` statement and then immediately loop through the rows of the result set. This statement can use either an implicit or explicit cursor.

If you use the `SELECT` statement only in the cursor `FOR LOOP` statement, then specify the `SELECT` statement inside the cursor `FOR LOOP` statement, as in Example 6–18. This form of the cursor `FOR LOOP` statement uses an implicit cursor, and is called an **implicit cursor `FOR LOOP` statement**. Because the implicit cursor is internal to the statement, you cannot reference it with the name `SQL`.

If you use the `SELECT` statement multiple times in the same PL/SQL unit, then define an explicit cursor for it and specify that cursor in the cursor `FOR LOOP` statement, as in Example 6–19. This form of the cursor `FOR LOOP` statement is called an **explicit cursor `FOR LOOP` statement**. You can use the same explicit cursor elsewhere in the same PL/SQL unit.

The cursor `FOR LOOP` statement implicitly declares its loop index as a `%ROWTYPE` record variable of the type that its cursor returns. This record is local to the loop and exists only during loop execution. Statements inside the loop can reference the record and its fields. They can reference virtual columns only by aliases, as in Example 6–21.

After declaring the loop index record variable, the `FOR LOOP` statement opens the specified cursor. With each iteration of the loop, the `FOR LOOP` statement fetches a row from the result set and stores it in the record. When there are no more rows to fetch, the cursor `FOR LOOP` statement closes the cursor. The cursor also closes if a statement inside the loop transfers control outside the loop or if PL/SQL raises an exception.

> **See Also:** "Cursor FOR LOOP Statement" on page 13-40 for its complete syntax and semantics

In Example 6–18, an implicit cursor `FOR LOOP` statement prints the last name and job ID of every clerk whose manager has an ID greater than 120.

Example 6–18 Implicit Cursor FOR LOOP Statement

```
BEGIN
  FOR item IN (
    SELECT last_name, job_id
    FROM employees
    WHERE job_id LIKE '%CLERK%'
    AND manager_id > 120
    ORDER BY last_name
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

```
Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
```

```
...
Name = Walsh, Job = SH_CLERK
```

Example 6–19 is like Example 6–18, except that it uses an explicit cursor `FOR LOOP` statement.

Example 6–19 Explicit Cursor FOR LOOP Statement

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE job_id LIKE '%CLERK%' AND manager_id > 120
    ORDER BY last_name;
BEGIN
  FOR item IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

```
Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
...
Name = Walsh, Job = SH_CLERK
```

Example 6–20 declares and defines an explicit cursor that accepts two parameters, and then uses it in an explicit cursor `FOR LOOP` statement to display the wages paid to employees who earn more than a specified wage in a specified department.

Example 6–20 Passing Parameters to Explicit Cursor FOR LOOP Statement

```
DECLARE
  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees
    WHERE job_id = job
    AND salary > max_wage;
BEGIN
  FOR person IN c1('ST_CLERK', 3000)
  LOOP
     -- process data record
    DBMS_OUTPUT.PUT_LINE (
      'Name = ' || person.last_name || ', salary = ' ||
      person.salary || ', Job Id = ' || person.job_id
    );
  END LOOP;
END;
/
```

Result:

```
Name = Nayer, salary = 4065.6, Job Id = ST_CLERK
Name = Mikkilineni, salary = 3430.35, Job Id = ST_CLERK
Name = Landry, salary = 3049.2, Job Id = ST_CLERK
...
Name = Vargas, salary = 3176.25, Job Id = ST_CLERK
```

In Example 6–21, the implicit cursor FOR LOOP references virtual columns by their aliases, full_name and dream_salary.

Example 6–21 Cursor FOR Loop References Virtual Columns

```
BEGIN
  FOR item IN (
    SELECT first_name || ' ' || last_name AS full_name,
           salary * 10                   AS dream_salary
    FROM employees
    WHERE ROWNUM <= 5
    ORDER BY dream_salary DESC, last_name ASC
  ) LOOP
    DBMS_OUTPUT.PUT_LINE
      (item.full_name || ' dreams of making ' || item.dream_salary);
  END LOOP;
END;
/
```

Result:

```
Michael Hartstein dreams of making 143325
Pat Fay dreams of making 66150
Jennifer Whalen dreams of making 48510
Douglas Grant dreams of making 31531.5
Donald OConnell dreams of making 31531.5
```

> **Note:** When an exception is raised inside a cursor FOR LOOP statement, the cursor closes before the exception handler runs. Therefore, the values of explicit cursor attributes are not available in the handler.

Query Result Set Processing With Explicit Cursors, OPEN, FETCH, and CLOSE

For full control over query result set processing, declare explicit cursors and manage them with the statements OPEN, FETCH, and CLOSE. (For instructions and examples, see "Explicit Cursors" on page 6-8.)

This result set processing technique is more complicated than the others, but it is also more flexible. For example, you can:

- Process multiple result sets in parallel, using multiple cursors.
- Process multiple rows in a single loop iteration, skip rows, or split the processing into multiple loops.
- Specify the query in one PL/SQL unit but retrieve the rows in another.

Query Result Set Processing with Subqueries

If you process a query result set by looping through it and running another query for each row, then you can improve performance by removing the second query from inside the loop and making it a subquery of the first query. For more information about subqueries, see *Oracle Database SQL Language Reference*.

Example 6–22 defines explicit cursor c1 with a query whose FROM clause contains a subquery.

Example 6–22 Subquery in FROM Clause of Parent Query

```
DECLARE
  CURSOR c1 IS
    SELECT t1.department_id, department_name, staff
    FROM departments t1,
         ( SELECT department_id, COUNT(*) AS staff
           FROM employees
           GROUP BY department_id
         ) t2
    WHERE (t1.department_id = t2.department_id) AND staff >= 5
    ORDER BY staff;

BEGIN
   FOR dept IN c1
   LOOP
     DBMS_OUTPUT.PUT_LINE ('Department = '
       || dept.department_name || ', staff = ' || dept.staff);
   END LOOP;
END;
/
```

Result:

```
Department = IT, staff = 5
Department = Purchasing, staff = 6
Department = Finance, staff = 6
Department = Sales, staff = 34
Department = Shipping, staff = 45
```

While an ordinary subquery is evaluated for each table, a **correlated subquery** is evaluated for each row. Example 6–23 returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the table, the correlated subquery computes the average salary for the corresponding department.

Example 6–23 Correlated Subquery

```
DECLARE
  CURSOR c1 IS
    SELECT department_id, last_name, salary
    FROM employees t
    WHERE salary > ( SELECT AVG(salary)
                     FROM employees
                     WHERE t.department_id = department_id
                   )
    ORDER BY department_id, last_name;
BEGIN
  FOR person IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE('Making above-average salary = ' || person.last_name);
  END LOOP;
END;
/
```

Result:

```
Making above-average salary = Hartstein
Making above-average salary = Raphaely
Making above-average salary = Bell
...
Making above-average salary = Higgins
```

Cursor Variables

A **cursor variable** is like an explicit cursor, except that:

- It is not limited to one query.

  You can open a cursor variable for a query, process the result set, and then use the cursor variable for another query.
- You can assign a value to it.
- You can use it in an expression.
- It can be a subprogram parameter.

  You can use cursor variables to pass query result sets between subprograms.
- It can be a host variable.

  You can use cursor variables to pass query result sets between PL/SQL stored subprograms and their clients.
- It cannot accept parameters.

  You cannot pass parameters to a cursor variable, but you can pass whole queries to it.

A cursor variable has this flexibility because it is a pointer; that is, its value is the address of an item, not the item itself.

Before you can reference a cursor variable, you must make it point to a SQL work area, either by opening it or by assigning it the value of an open PL/SQL cursor variable or open host cursor variable.

Note: Cursor variables and explicit cursors are not interchangeable—you cannot use one where the other is expected. For example, you cannot reference a cursor variable in a cursor `FOR LOOP` statement.

Topics

See Also: "Restrictions on Cursor Variables" on page 13-42

Creating Cursor Variables

To create a cursor variable, either declare a variable of the predefined type `SYS_REFCURSOR` or define a `REF CURSOR` type and then declare a variable of that type.

Note: Informally, a cursor variable is sometimes called a REF CURSOR).

The basic syntax of a REF CURSOR type definition is:

```
TYPE type_name IS REF CURSOR [ RETURN return_type ]
```

(For the complete syntax and semantics, see "Cursor Variable Declaration" on page 13-42.)

If you specify *return_type*, then the REF CURSOR type and cursor variables of that type are **strong**; if not, they are **weak**. SYS_REFCURSOR and cursor variables of that type are weak.

With a strong cursor variable, you can associate only queries that return the specified type. With a weak cursor variable, you can associate any query.

Weak cursor variables are more error-prone than strong ones, but they are also more flexible. Weak REF CURSOR types are interchangeable with each other and with the predefined type SYS_REFCURSOR. You can assign the value of a weak cursor variable to any other weak cursor variable.

You can assign the value of a strong cursor variable to another strong cursor variable only if both cursor variables have the same type (not merely the same return type).

Note: You can partition weak cursor variable arguments to table functions only with the PARTITION BY ANY clause, not with PARTITION BY RANGE or PARTITION BY HASH. For syntax and semantics, see "parallel_enable_clause ::=" on page 14-34 and "parallel_enable_clause" on page 14-36.

Example 6–24 defines strong and weak REF CURSOR types, variables of those types, and a variable of the predefined type SYS_REFCURSOR.

Example 6–24 Cursor Variable Declarations

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;  -- strong type
  TYPE genericcurtyp IS REF CURSOR;                       -- weak type

  cursor1  empcurtyp;       -- strong cursor variable
  cursor2  genericcurtyp;   -- weak cursor variable
  my_cursor SYS_REFCURSOR;  -- weak cursor variable

  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE;  -- strong type
  dept_cv deptcurtyp;  -- strong cursor variable
BEGIN
  NULL;
END;
/
```

In Example 6–25, *return_type* is a user-defined RECORD type.

Example 6–25 Cursor Variable with User-Defined Return Type

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    employee_id NUMBER,
```

```
    last_name VARCHAR2(25),
    salary   NUMBER(8,2));

  TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
  emp_cv EmpCurTyp;
BEGIN
  NULL;
END;
/
```

Opening and Closing Cursor Variables

After declaring a cursor variable, you can open it with the `OPEN FOR` statement, which does the following:

1. Associates the cursor variable with a query (typically, the query returns multiple rows)

   The query can include placeholders for bind variables, whose values you specify in the `USING` clause of the `OPEN FOR` statement.

2. Allocates database resources to process the query
3. Processes the query; that is:
   1. Identifies the result set

      If the query references variables, their values affect the result set. For details, see "Variables in Cursor Variable Queries" on page 6-33.

   2. If the query has a `FOR UPDATE` clause, locks the rows of the result set

      For details, see "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-46.

4. Positions the cursor before the first row of the result set

You need not close a cursor variable before reopening it (that is, using it in another `OPEN FOR` statement). After you reopen a cursor variable, the query previously associated with it is lost.

When you no longer need a cursor variable, close it with the `CLOSE` statement, thereby allowing its resources to be reused. After closing a cursor variable, you cannot fetch records from its result set or reference its attributes. If you try, PL/SQL raises the predefined exception `INVALID_CURSOR`.

You can reopen a closed cursor variable.

See Also:

- "OPEN FOR Statement" on page 13-104 for its syntax and semantics
- "CLOSE Statement" on page 13-23 for its syntax and semantics

Fetching Data with Cursor Variables

After opening a cursor variable, you can fetch the rows of the query result set with the `FETCH` statement, which works as described in "Fetching Data with Explicit Cursors" on page 6-10.

The return type of the cursor variable must be compatible with the *`into_clause`* of the `FETCH` statement. If the cursor variable is strong, PL/SQL catches incompatibility at

compile time. If the cursor variable is weak, PL/SQL catches incompatibility at run time, raising the predefined exception ROWTYPE_MISMATCH before the first fetch.

Example 6–26 uses one cursor variable to do what Example 6–6 does with two explicit cursors. The first OPEN FOR statement includes the query itself. The second OPEN FOR statement references a variable whose value is a query.

Example 6–26 Fetching Data with Cursor Variables

```
DECLARE
  cv SYS_REFCURSOR;  -- cursor variable

  v_lastname  employees.last_name%TYPE;  -- variable for last_name
  v_jobid     employees.job_id%TYPE;     -- variable for job_id

  query_2 VARCHAR2(200) :=
    'SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, ''[ACADFIMKSA]_M[ANGR]'')
    ORDER BY job_id';

  v_employees employees%ROWTYPE;  -- record variable row of table

BEGIN
  OPEN cv FOR
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;

  LOOP  -- Fetches 2 columns into variables
    FETCH cv INTO v_lastname, v_jobid;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( '-----------------------------------' );

  OPEN cv FOR query_2;

  LOOP  -- Fetches entire row into the v_employees record
    FETCH cv INTO v_employees;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_employees.last_name, 25, ' ') ||
                          v_employees.job_id );
  END LOOP;

  CLOSE cv;
END;
/
```

Result:

```
Atkinson                  ST_CLERK
Bell                      SH_CLERK
Bissot                    ST_CLERK
...
Walsh                     SH_CLERK
-----------------------------------
Higgins                   AC_MGR
Greenberg                 FI_MGR
Hartstein                 MK_MAN
...
```

```
Zlotkey                   SA_MAN
```

Example 6–27 fetches from a cursor variable into two collections (nested tables), using the `BULK COLLECT` clause of the `FETCH` statement.

Example 6–27 Fetching from Cursor Variable into Collections

```
DECLARE
  TYPE empcurtyp IS REF CURSOR;
  TYPE namelist IS TABLE OF employees.last_name%TYPE;
  TYPE sallist IS TABLE OF employees.salary%TYPE;
  emp_cv  empcurtyp;
  names   namelist;
  sals    sallist;
BEGIN
  OPEN emp_cv FOR
    SELECT last_name, salary FROM employees
    WHERE job_id = 'SA_REP'
    ORDER BY salary DESC;

  FETCH emp_cv BULK COLLECT INTO names, sals;
  CLOSE emp_cv;
  -- loop through the names and sals collections
  FOR i IN names.FIRST .. names.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || names(i) || ', salary = ' || sals(i));
  END LOOP;
END;
/
```

Result:

```
Name = Ozer, salary = 12075
Name = Abel, salary = 11550
Name = Vishney, salary = 11025
...
Name = Kumar, salary = 6405
```

See Also:

- "FETCH Statement" on page 13-71 for its complete syntax and semantics
- "FETCH Statement with BULK COLLECT Clause" on page 12-31 for information about `FETCH` statements that return more than one row at a time

Assigning Values to Cursor Variables

You can assign to a PL/SQL cursor variable the value of another PL/SQL cursor variable or host cursor variable. The syntax is:

```
target_cursor_variable := source_cursor_variable;
```

If *source_cursor_variable* is open, then after the assignment, *target_cursor_variable* is also open. The two cursor variables point to the same SQL work area.

If *source_cursor_variable* is not open, opening *target_cursor_variable* after the assignment does not open *source_cursor_variable*.

Variables in Cursor Variable Queries

The query associated with a cursor variable can reference any variable in its scope. When you open a cursor variable with the `OPEN FOR` statement, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

Example 6–28 opens a cursor variable for a query that references the variable `factor`, which has the value 2. Therefore, `sal_multiple` is always 2 times `sal`, despite that `factor` is incremented after every fetch.

Example 6–28 Variable in Cursor Variable Query—No Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  cv SYS_REFCURSOR;

BEGIN
  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;

  CLOSE cv;
END;
/
```

Result:

```
factor = 2
sal          = 4451
sal_multiple = 8902
factor = 3
sal          = 26460
sal_multiple = 52920
factor = 4
sal          = 18742.5
sal_multiple = 37485
factor = 5
sal          = 18742.5
sal_multiple = 37485
```

To change the result set, you must change the value of the variable and then open the cursor variable again for the same query, as in Example 6–29.

Example 6–29 Variable in Cursor Variable Query—Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
```

```
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  cv SYS_REFCURSOR;

BEGIN
  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);

  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;

  factor := factor + 1;

  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);

  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;

  CLOSE cv;
END;
/
```

Result:

```
factor = 2
sal          = 4451
sal_multiple = 8902
sal          = 26460
sal_multiple = 52920
sal          = 18742.5
sal_multiple = 37485
sal          = 18742.5
sal_multiple = 37485
factor = 3
sal          = 4451
sal_multiple = 13353
sal          = 26460
sal_multiple = 79380
sal          = 18742.5
sal_multiple = 56227.5
sal          = 18742.5
sal_multiple = 56227.5
```

Cursor Variable Attributes

A cursor variable has the same attributes as an explicit cursor (see "Explicit Cursor Attributes" on page 6-19). The syntax for the value of a cursor variable attribute is *cursor_variable_name* immediately followed by *attribute* (for example, `cv%ISOPEN`). If a cursor variable is not open, referencing any attribute except `%ISOPEN` raises the predefined exception `INVALID_CURSOR`.

Cursor Variables as Subprogram Parameters

You can use a cursor variable as a subprogram parameter, which makes it useful for passing query results between subprograms. For example:

- You can open a cursor variable in one subprogram and process it in a different subprogram.
- In a multilanguage application, a PL/SQL subprogram can use a cursor variable to return a result set to a subprogram written in a different language.

> **Note:** The invoking and invoked subprograms must be in the same database instance. You cannot pass or return cursor variables to subprograms invoked through database links.

> **Caution:** Because cursor variables are pointers, using them as subprogram parameters increases the likelihood of subprogram parameter aliasing, which can have unintended results. For more information, see "Subprogram Parameter Aliasing with Cursor Variable Parameters" on page 8-20.

When declaring a cursor variable as the formal parameter of a subprogram:

- If the subprogram opens or assigns a value to the cursor variable, then the parameter mode must be `IN OUT`.
- If the subprogram only fetches from, or closes, the cursor variable, then the parameter mode can be either `IN` or `IN OUT`.

Corresponding formal and actual cursor variable parameters must have compatible return types. Otherwise, PL/SQL raises the predefined exception `ROWTYPE_MISMATCH`.

To pass a cursor variable parameter between subprograms in different PL/SQL units, define the `REF CURSOR` type of the parameter in a package. When the type is in a package, multiple subprograms can use it. One subprogram can declare a formal parameter of that type, and other subprograms can declare variables of that type and pass them to the first subprogram.

Example 6–30 defines, in a package, a `REF CURSOR` type and a procedure that opens a cursor variable parameter of that type.

Example 6–30 Procedure to Open Cursor Variable for One Query

```
CREATE OR REPLACE PACKAGE emp_data AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp);
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
```

```
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/
```

In Example 6–31,the stored procedure opens its cursor variable parameter for a chosen query. The queries have the same return type.

Example 6–31 Opening Cursor Variable for Chosen Query (Same Return Type)

```
CREATE OR REPLACE PACKAGE emp_data AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT);
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE commission_pct IS NOT NULL;
    ELSIF choice = 2 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE salary > 2500;
    ELSIF choice = 3 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE department_id = 100;
    END IF;
  END;
END emp_data;
/
```

In Example 6–32,the stored procedure opens its cursor variable parameter for a chosen query. The queries have the different return types.

Example 6–32 Opening Cursor Variable for Chosen Query (Different Return Types)

```
CREATE OR REPLACE PACKAGE admin_data AS
  TYPE gencurtyp IS REF CURSOR;
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT);
END admin_data;
/
CREATE OR REPLACE PACKAGE BODY admin_data AS
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN generic_cv FOR SELECT * FROM employees;
    ELSIF choice = 2 THEN
      OPEN generic_cv FOR SELECT * FROM departments;
    ELSIF choice = 3 THEN
      OPEN generic_cv FOR SELECT * FROM jobs;
    END IF;
  END;
END admin_data;
/
```

See Also:

- "Subprogram Parameters" on page 8-9 for more information about subprogram parameters
- "CURSOR Expressions" on page 6-38 for information about `CURSOR` expressions, which can be actual parameters for formal cursor variable parameters
- Chapter 10, "PL/SQL Packages," for more information about packages

Cursor Variables as Host Variables

You can use a cursor variable as a host variable, which makes it useful for passing query results between PL/SQL stored subprograms and their clients. When a cursor variable is a host variable, PL/SQL and the client (the host environment) share a pointer to the SQL work area that stores the result set.

To use a cursor variable as a host variable, declare the cursor variable in the host environment and then pass it as an input host variable (bind variable) to PL/SQL. Host cursor variables are compatible with any query return type (like weak PL/SQL cursor variables).

In Example 6–33, a Pro*C client program declares a cursor variable and a selector and passes them as host variables to a PL/SQL anonymous block, which opens the cursor variable for the selected query.

Example 6–33 Cursor Variable as Host Variable in Pro*C Client Program

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL_CURSOR  generic_cv;  -- Declare host cursor variable.
  int         choice;      -- Declare selector.
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :generic_cv;  -- Initialize host cursor variable.
-- Pass host cursor variable and selector to PL/SQL block.
/
EXEC SQL EXECUTE
BEGIN
  IF :choice = 1 THEN
    OPEN :generic_cv FOR SELECT * FROM employees;
  ELSIF :choice = 2 THEN
    OPEN :generic_cv FOR SELECT * FROM departments;
  ELSIF :choice = 3 THEN
    OPEN :generic_cv FOR SELECT * FROM jobs;
  END IF;
END;
END-EXEC;
```

A SQL work area remains accessible while any cursor variable points to it, even if you pass the value of a cursor variable from one scope to another. For example, in Example 6–33, the Pro*C program passes a host cursor variable to an embedded PL/SQL anonymous block. After the block runs, the cursor variable still points to the SQL work area.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, and continue to fetch from it on the client side. You can also reduce network traffic with a PL/SQL anonymous block that opens or closes several host cursor variables in a single round trip. For example:

```
/* PL/SQL anonymous block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM employees;
  OPEN :dept_cv FOR SELECT * FROM departments;
  OPEN :loc_cv FOR SELECT * FROM locations;
END;
/
```

Because the cursor variables still point to the SQL work areas after the PL/SQL anonymous block runs, the client program can use them. When the client program no longer needs the cursors, it can use a PL/SQL anonymous block to close them. For example:

```
/* PL/SQL anonymous block in host environment */
BEGIN
  CLOSE :emp_cv;
  CLOSE :dept_cv;
  CLOSE :loc_cv;
END;
/
```

This technique is useful for populating a multiblock form, as in Oracle Forms. For example, you can open several SQL work areas in a single round trip, like this:

```
/* PL/SQL anonymous block in host environment */
BEGIN
  OPEN :c1 FOR SELECT 1 FROM DUAL;
  OPEN :c2 FOR SELECT 1 FROM DUAL;
  OPEN :c3 FOR SELECT 1 FROM DUAL;
END;
/
```

Note: If you bind a host cursor variable into PL/SQL from an Oracle Call Interface (OCI) client, then you cannot fetch from it on the server side unless you also open it there on the same server call.

CURSOR Expressions

A `CURSOR` expression returns a nested cursor. It has this syntax:

```
CURSOR ( subquery )
```

You can use a `CURSOR` expression in a `SELECT` statement that is not a subquery (as in Example 6–34) or pass it to a function that accepts a cursor variable parameter (see "Passing CURSOR Expressions to Pipelined Table Functions" on page 12-44). You cannot use a cursor expression with an implicit cursor.

> **See Also:** *Oracle Database SQL Language Reference* for more information about `CURSOR` expressions, including restrictions

Example 6–34 declares and defines an explicit cursor for a query that includes a cursor expression. For each department in the `departments` table, the nested cursor returns the last name of each employee in that department (which it retrieves from the `employees` table).

Example 6–34 CURSOR Expression

```
DECLARE
  TYPE emp_cur_typ IS REF CURSOR;
```

```
  emp_cur    emp_cur_typ;
  dept_name  departments.department_name%TYPE;
  emp_name   employees.last_name%TYPE;

  CURSOR c1 IS
    SELECT department_name,
      CURSOR ( SELECT e.last_name
                 FROM employees e
                 WHERE e.department_id = d.department_id
                 ORDER BY e.last_name
               ) employees
    FROM departments d
    WHERE department_name LIKE 'A%'
    ORDER BY department_name;
BEGIN
  OPEN c1;
  LOOP  -- Process each row of query result set
    FETCH c1 INTO dept_name, emp_cur;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Department: ' || dept_name);

    LOOP -- Process each row of subquery result set
      FETCH emp_cur INTO emp_name;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('-- Employee: ' || emp_name);
    END LOOP;
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
Department: Accounting
-- Employee: Gietz
-- Employee: Higgins
Department: Administration
-- Employee: Whalen
```

Transaction Processing and Control

A **transaction** is a sequence of one or more SQL statements that Oracle Database treats as a unit: either all of the statements are performed, or none of them are. For more information about transactions, see *Oracle Database Concepts*.

Transaction processing is an Oracle Database feature that lets multiple users work on the database concurrently, and ensures that each user sees a consistent version of data and that all changes are applied in the right order. For more information about transaction processing, see *Oracle Database Concepts*.

Different users can write to the same data structures without harming each other's data or coordinating with each other, because Oracle Database locks data structures automatically. To maximize data availability, Oracle Database locks the minimum amount of data for the minimum amount of time. For more information about the Oracle Database locking mechanism, see *Oracle Database Concepts*.

You rarely must write extra code to prevent problems with multiple users accessing data concurrently. However, if you do need this level of control, you can manually

override the Oracle Database default locking mechanisms. For more information about manual data locks, see *Oracle Database Concepts*.

Topics

COMMIT Statement

The `COMMIT` statement ends the current transaction, making its changes permanent and visible to other users.

> **Note:** A transaction can span multiple blocks, and a block can contain multiple transactions.

The `WRITE` clause of the `COMMIT` statement specifies the priority with which Oracle Database writes to the redo log the information that the commit operation generates.

In Example 6–35, a transaction transfers money from one bank account to another. It is important that the money both leaves one account and enters the other, hence the `COMMIT WRITE IMMEDIATE NOWAIT` statement.

Example 6–35 COMMIT Statement with COMMENT and WRITE Clauses

```
DROP TABLE accounts;
CREATE TABLE accounts (
  account_id  NUMBER(6),
  balance     NUMBER (10,2)
);

INSERT INTO accounts (account_id, balance)
VALUES (7715, 6350.00);

INSERT INTO accounts (account_id, balance)
VALUES (7720, 5100.50);

CREATE OR REPLACE PROCEDURE transfer (
  from_acct  NUMBER,
  to_acct    NUMBER,
  amount     NUMBER
) AUTHID DEFINER AS
BEGIN
  UPDATE accounts
  SET balance = balance - amount
  WHERE account_id = from_acct;

  UPDATE accounts
  SET balance = balance + amount
  WHERE account_id = to_acct;
```

```
  COMMIT WRITE IMMEDIATE NOWAIT;
END;
/
```

Query before transfer:

```
SELECT * FROM accounts;
```

Result:

```
ACCOUNT_ID    BALANCE
---------- ----------
      7715       6350
      7720     5100.5

BEGIN
  transfer(7715, 7720, 250);
END;
/
```

Query after transfer:

```
SELECT * FROM accounts;
```

Result:

```
ACCOUNT_ID    BALANCE
---------- ----------
      7715       6100
      7720     5350.5
```

Note: The default PL/SQL commit behavior for nondistributed transactions is `BATCH NOWAIT` if the `COMMIT_LOGGING` and `COMMIT_WAIT` database initialization parameters have not been set.

See Also:

- *Oracle Database Advanced Application Developer's Guide* for more information about committing transactions
- *Oracle Database Concepts* for information about distributed transactions
- *Oracle Database SQL Language Reference*for information about the `COMMIT` statement
- *Oracle Data Guard Concepts and Administration* for information about ensuring no loss of data during a failover to a standby database

ROLLBACK Statement

The `ROLLBACK` statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because a SQL statement fails or PL/SQL raises an exception, a rollback lets you take corrective action and perhaps start over.

Example 6–36 inserts information about an employee into three different tables. If an `INSERT` statement tries to store a duplicate employee number, PL/SQL raises the

predefined exception DUP_VAL_ON_INDEX. To ensure that changes to all three tables are undone, the exception handler runs a ROLLBACK.

Example 6–36 ROLLBACK Statement

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);


DROP TABLE emp_sal;
CREATE TABLE emp_sal AS
  SELECT employee_id, salary
  FROM employees;

CREATE UNIQUE INDEX empsal_ix
ON emp_sal (employee_id);


DROP TABLE emp_job;
CREATE TABLE emp_job AS
  SELECT employee_id, job_id
  FROM employees;

CREATE UNIQUE INDEX empjobid_ix
ON emp_job (employee_id);


DECLARE
  emp_id        NUMBER(6);
  emp_lastname  VARCHAR2(25);
  emp_salary    NUMBER(8,2);
  emp_jobid     VARCHAR2(10);
BEGIN
  SELECT employee_id, last_name, salary, job_id
  INTO emp_id, emp_lastname, emp_salary, emp_jobid
  FROM employees
  WHERE employee_id = 120;

  INSERT INTO emp_name (employee_id, last_name)
  VALUES (emp_id, emp_lastname);

  INSERT INTO emp_sal (employee_id, salary)
  VALUES (emp_id, emp_salary);

  INSERT INTO emp_job (employee_id, job_id)
  VALUES (emp_id, emp_jobid);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Inserts were rolled back');
END;
/
```

See Also: *Oracle Database SQL Language Reference* for more information about the ROLLBACK statement

SAVEPOINT Statement

The SAVEPOINT statement names and marks the current point in the processing of a transaction. Savepoints let you roll back part of a transaction instead of the whole transaction. The number of active savepoints for each session is unlimited.

Example 6–37 marks a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the employee_id column, PL/SQL raises the predefined exception DUP_VAL_ON_INDEX and the transaction rolls back to the savepoint, undoing only the INSERT statement.

Example 6–37 SAVEPOINT and ROLLBACK Statements

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name, salary
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
  emp_id        employees.employee_id%TYPE;
  emp_lastname  employees.last_name%TYPE;
  emp_salary    employees.salary%TYPE;

BEGIN
  SELECT employee_id, last_name, salary
  INTO emp_id, emp_lastname, emp_salary
  FROM employees
  WHERE employee_id = 120;

  UPDATE emp_name
  SET salary = salary * 1.1
  WHERE employee_id = emp_id;

  DELETE FROM emp_name
  WHERE employee_id = 130;

  SAVEPOINT do_insert;

  INSERT INTO emp_name (employee_id, last_name, salary)
  VALUES (emp_id, emp_lastname, emp_salary);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO do_insert;
  DBMS_OUTPUT.PUT_LINE('Insert was rolled back');
END;
/
```

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints.

If you mark a savepoint in a recursive subprogram, new instances of the SAVEPOINT statement run at each level in the recursive descent, but you can only roll back to the most recently marked savepoint.

Savepoint names are undeclared identifiers. Reusing a savepoint name in a transaction moves the savepoint from its old position to the current point in the transaction, which means that a rollback to the savepoint affects only the current part of the transaction.

Example 6–38 Reusing SAVEPOINT with ROLLBACK

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name, salary
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
  emp_id        employees.employee_id%TYPE;
  emp_lastname  employees.last_name%TYPE;
  emp_salary    employees.salary%TYPE;

BEGIN
  SELECT employee_id, last_name, salary
  INTO emp_id, emp_lastname, emp_salary
  FROM employees
  WHERE employee_id = 120;

  SAVEPOINT my_savepoint;

  UPDATE emp_name
  SET salary = salary * 1.1
  WHERE employee_id = emp_id;

  DELETE FROM emp_name
  WHERE employee_id = 130;

  SAVEPOINT my_savepoint;

  INSERT INTO emp_name (employee_id, last_name, salary)
  VALUES (emp_id, emp_lastname, emp_salary);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO my_savepoint;
    DBMS_OUTPUT.PUT_LINE('Transaction rolled back.');
END;
/
```

> **See Also:** *Oracle Database SQL Language Reference* for more information about the SET TRANSACTION SQL statement

Implicit Rollbacks

Before running an INSERT, UPDATE, DELETE, or MERGE statement, the database marks an implicit savepoint (unavailable to you). If the statement fails, the database rolls back to the savepoint. Usually, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back. (For information about handling exceptions, see

Chapter 11, "PL/SQL Error Handling").

The database can also roll back single SQL statements to break deadlocks. The database signals an error to a participating transaction and rolls back the current statement in that transaction.

Before running a SQL statement, the database must parse it, that is, examine it to ensure it follows syntax rules and refers to valid schema objects. Errors detected while running a SQL statement cause a rollback, but errors detected while parsing the statement do not.

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to `OUT` parameters, and does not do any rollback.

SET TRANSACTION Statement

You use the `SET TRANSACTION` statement to begin a read-only or read-write transaction, establish an isolation level, or assign your current transaction to a specified rollback segment. Read-only transactions are useful for running multiple queries while other users update the same tables.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multi-table, multi-query, read-consistent view. Other users can continue to query or update data as usual. A commit or rollback ends the transaction.

In Example 6–39 a read-only transaction gather order totals for the day, the past week, and the past month. The totals are unaffected by other users updating the database during the transaction. The `orders` table is in the sample schema OE.

Example 6–39 SET TRANSACTION Statement in Read-Only Transaction

```
DECLARE
  daily_order_total    NUMBER(12,2);
  weekly_order_total   NUMBER(12,2);
  monthly_order_total  NUMBER(12,2);
BEGIN
   COMMIT; -- end previous transaction
   SET TRANSACTION READ ONLY NAME 'Calculate Order Totals';

   SELECT SUM (order_total)
   INTO daily_order_total
   FROM orders
   WHERE order_date = SYSDATE;

   SELECT SUM (order_total)
   INTO weekly_order_total
   FROM orders
   WHERE order_date = SYSDATE - 7;

   SELECT SUM (order_total)
   INTO monthly_order_total
   FROM orders
   WHERE order_date = SYSDATE - 30;

   COMMIT; -- ends read-only transaction
END;
/
```

The `SET TRANSACTION` statement must be the first SQL statement in a read-only transaction and can appear only once in a transaction. If you set a transaction to `READ`

ONLY, subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

Only the SELECT, OPEN, FETCH, CLOSE, LOCK TABLE, COMMIT, and ROLLBACK statements are allowed in a read-only transaction. Queries cannot be FOR UPDATE.

> **See Also:** *Oracle Database SQL Language Reference* for more information about the SQL statement SET TRANSACTION

Overriding Default Locking

By default, Oracle Database locks data structures automatically, which lets different applications write to the same data structures without harming each other's data or coordinating with each other.

If you must have exclusive access to data during a transaction, you can override default locking with these SQL statements:

- LOCK TABLE, which explicitly locks entire tables.
- SELECT with the FOR UPDATE clause (SELECT FOR UPDATE) , which explicitly locks specific rows of a table.

Topics

- LOCK TABLE Statement
- SELECT FOR UPDATE and FOR UPDATE Cursors
- Simulating CURRENT OF Clause with ROWID Pseudocolumn

LOCK TABLE Statement

The LOCK TABLE statement explicitly locks one or more tables in a specified lock mode so that you can share or deny access to them.

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an exclusive lock. While one user has an exclusive lock on a table, no other users can insert, delete, or update rows in that table.

A table lock never prevents other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row does one transaction wait for the other to complete. The LOCK TABLE statement lets you specify how long to wait for another transaction to complete.

Table locks are released when the transaction that acquired them is either committed or rolled back.

> **See Also:**
>
> - *Oracle Database Advanced Application Developer's Guide* for more information about locking tables explicitly
> - *Oracle Database SQL Language Reference* for more information about the LOCK TABLE statement

SELECT FOR UPDATE and FOR UPDATE Cursors

The SELECT statement with the FOR UPDATE clause (SELECT FOR UPDATE statement) selects the rows of the result set and locks them. SELECT FOR UPDATE lets you base an update on the existing values in the rows, because it ensures that no other user can change those values before you update them. You can also use SELECT FOR UPDATE to

lock rows that you do not want to update, as in Example 9–11.

> **Note:** In tables compressed with Hybrid Columnar Compression (HCC), DML statements lock compression units rather than rows. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

By default, the SELECT FOR UPDATE statement waits until the requested row lock is acquired. To change this behavior, use the NOWAIT, WAIT, or SKIP LOCKED clause of the SELECT FOR UPDATE statement. For information about these clauses, see *Oracle Database SQL Language Reference*.

When SELECT FOR UPDATE is associated with an explicit cursor, the cursor is called a **FOR UPDATE cursor**. Only a FOR UPDATE cursor can appear in the CURRENT OF clause of an UPDATE or DELETE statement. (The CURRENT OF clause, a PL/SQL extension to the WHERE clause of the SQL statements UPDATE and DELETE, restricts the statement to the current row of the cursor.)

In Example 6–40, a FOR UPDATE cursor appears in the CURRENT OF clause of an UPDATE statement.

Example 6–40 FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement

```
DECLARE
  my_emp_id NUMBER(6);
  my_job_id VARCHAR2(10);
  my_sal    NUMBER(8,2);
  CURSOR c1 IS
    SELECT employee_id, job_id, salary
    FROM employees FOR UPDATE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_emp_id, my_job_id, my_sal;
    IF my_job_id = 'SA_REP' THEN
      UPDATE employees
      SET salary = salary * 1.02
      WHERE CURRENT OF c1;
    END IF;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
END;
/
```

When SELECT FOR UPDATE queries multiple tables, it locks only rows whose columns appear in the FOR UPDATE clause.

In Example 6–41, SELECT FOR UPDATE queries the tables EMPLOYEES and DEPARTMENTS, but only SALARY appears in the FOR UPDATE clause. SALARY is a column of EMPLOYEES, but not of DEPARTMENTS; therefore, SELECT FOR UPDATE locks only rows of EMPLOYEES. If the FOR UPDATE clause included DEPARTMENT_ID or MANAGER_ID, which are columns of both EMPLOYEES and DEPARTMENTS, SELECT FOR UPDATE would lock rows of both tables.

Example 6–41 SELECT FOR UPDATE Statement for Multiple Tables

```
DECLARE
  CURSOR c1 IS
```

```
    SELECT last_name, department_name
    FROM employees, departments
    WHERE employees.department_id = departments.department_id
    AND job_id = 'SA_MAN'
    FOR UPDATE OF salary;
BEGIN
  NULL;
END;
/
```

Simulating CURRENT OF Clause with ROWID Pseudocolumn

The rows of the result set are locked when you open a `FOR UPDATE` cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction. After the rows are unlocked, you cannot fetch from the `FOR UPDATE` cursor, as Example 6–42 shows (the result is the same if you substitute `ROLLBACK` for `COMMIT`).

Example 6–42 FETCH with FOR UPDATE Cursor After COMMIT Statement

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  CURSOR c1 IS
    SELECT * FROM emp
    FOR UPDATE OF salary
    ORDER BY employee_id;

  emp_rec  emp%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;  -- fails on second iteration
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (
      'emp_rec.employee_id = ' ||
      TO_CHAR(emp_rec.employee_id)
    );

    UPDATE emp
    SET salary = salary * 1.05
    WHERE employee_id = 105;

    COMMIT;  -- releases locks
  END LOOP;
END;
/
```

Result:

```
emp_rec.employee_id = 100
DECLARE
*
ERROR at line 1:
ORA-01002: fetch out of sequence
ORA-06512: at line 11
```

The workaround is to simulate the `CURRENT OF` clause with the `ROWID` pseudocolumn (described in *Oracle Database SQL Language Reference*). Select the rowid of each row into a `UROWID` variable and use the rowid to identify the current row during subsequent

updates and deletes, as in Example 6–43. (To print the value of a UROWID variable, convert it to VARCHAR2, using the ROWIDTOCHAR function described in *Oracle Database SQL Language Reference*.)

Note: When you update a row in a table compressed with Hybrid Columnar Compression (HCC), the ROWID of the row changes. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

Caution: Because no FOR UPDATE clause locks the fetched rows, other users might unintentionally overwrite your changes.

Note: The extra space needed for read consistency is not released until the cursor is closed, which can slow down processing for large updates.

Example 6–43 Simulating CURRENT OF Clause with ROWID Pseudocolumn

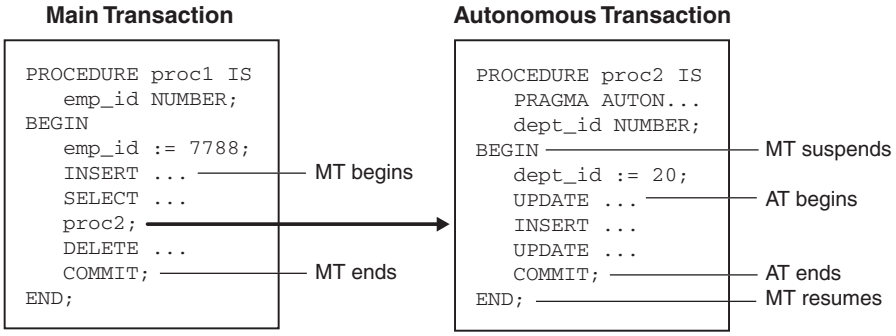
```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id, rowid
    FROM emp;  -- no FOR UPDATE clause

  my_lastname   employees.last_name%TYPE;
  my_jobid      employees.job_id%TYPE;
  my_rowid      UROWID;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_lastname, my_jobid, my_rowid;
    EXIT WHEN c1%NOTFOUND;

    UPDATE emp
    SET salary = salary * 1.02
    WHERE rowid = my_rowid;  -- simulates WHERE CURRENT OF c1

    COMMIT;
  END LOOP;
  CLOSE c1;
END;
/
```

Autonomous Transactions

An **autonomous transaction** is an independent transaction started by another transaction, the main transaction. Autonomous transactions do SQL operations and commit or roll back, without committing or rolling back the main transaction.

Figure 6–1 shows how control flows from the main transaction (MT) to an autonomous transaction (AT) and back again.

Figure 6–1 Transaction Control Flow



Note: Although an autonomous transaction is started by another transaction, it is not a nested transaction, because:

- It does not share transactional resources (such as locks) with the main transaction.
- It does not depend on the main transaction.

  For example, if the main transaction rolls back, nested transactions roll back, but autonomous transactions do not.
- Its committed changes are visible to other transactions immediately.

  A nested transaction's committed changes are not visible to other transactions until the main transaction commits.
- Exceptions raised in an autonomous transaction cause a transaction-level rollback, not a statement-level rollback.

Topics

Advantages of Autonomous Transactions

After starting, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

Autonomous transactions help you build modular, reusable software components. You can encapsulate autonomous transactions in stored subprograms. An invoking application needs not know whether operations done by that stored subprogram succeeded or failed.

Transaction Context

The main transaction shares its context with nested routines, but not with autonomous transactions. When one autonomous routine invokes another (or itself, recursively), the routines share no transaction context. When an autonomous routine invokes a nonautonomous routine, the routines share the same transaction context.

Transaction Visibility

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. These changes become visible to the main transaction when it resumes, if its isolation level is set to `READ COMMITTED` (the default).

If you set the isolation level of the main transaction to `SERIALIZABLE`, changes made by its autonomous transactions are *not* visible to the main transaction when it resumes:

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

Note:

- Transaction properties apply only to the transaction in which they are set.
- Cursor attributes are not affected by autonomous transactions.

Declaring Autonomous Transactions

To declare an autonomous transaction, use the `AUTONOMOUS_TRANSACTION` pragma. For information about this pragma, see "AUTONOMOUS_TRANSACTION Pragma" on page 13-6.

Tip: For readability, put the `AUTONOMOUS_TRANSACTION` pragma at the top of the declarative section. (The pragma is allowed anywhere in the declarative section.)

You cannot apply the `AUTONOMOUS_TRANSACTION` pragma to an entire package or ADT, but you can apply it to each subprogram in a package or each method of an ADT.

Example 6–44 marks a package function as autonomous.

Example 6–44 Declaring Autonomous Function in Package

```
CREATE OR REPLACE PACKAGE emp_actions AS  -- package specification
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
  RETURN NUMBER;
END emp_actions;
/
CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- package body
  -- code for function raise_salary
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
  RETURN NUMBER IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    new_sal NUMBER(8,2);
  BEGIN
    UPDATE employees SET salary =
      salary + sal_raise WHERE employee_id = emp_id;
    COMMIT;
    SELECT salary INTO new_sal FROM employees
      WHERE employee_id = emp_id;
```

```
    RETURN new_sal;
  END raise_salary;
END emp_actions;
/
```

Example 6–45 marks a standalone subprogram as autonomous.

Example 6–45 Declaring Autonomous Standalone Procedure

```
CREATE OR REPLACE PROCEDURE lower_salary
   (emp_id NUMBER, amount NUMBER)
AS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  UPDATE employees
  SET salary =  salary - amount
  WHERE employee_id = emp_id;

  COMMIT;
END lower_salary;
/
```

Example 6–46 marks a schema-level PL/SQL block as autonomous. (A nested PL/SQL block cannot be autonomous.)

Example 6–46 Declaring Autonomous PL/SQL Block

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  emp_id NUMBER(6)   := 200;
  amount NUMBER(6,2) := 200;
BEGIN
  UPDATE employees
  SET salary =  salary - amount
  WHERE employee_id = emp_id;

  COMMIT;
END;
/
```

Controlling Autonomous Transactions

The first SQL statement in an autonomous routine begins a transaction. When one transaction ends, the next SQL statement begins another transaction. All SQL statements run since the last commit or rollback comprise the current transaction. To control autonomous transactions, use these statements, which apply only to the current (active) transaction:

- COMMIT
- ROLLBACK [TO *savepoint_name*]
- SAVEPOINT *savepoint_name*
- SET TRANSACTION

Topics

Entering and Exiting

When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes.

If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception goes unhandled, or if the transaction ends because of some other unhandled exception, the transaction is rolled back.

To exit normally, you must explicitly commit or roll back all autonomous transactions. If the routine (or any routine invoked by it) has pending transactions, PL/SQL raises an exception and the pending transactions are rolled back.

Committing and Rolling Back

`COMMIT` and `ROLLBACK` end the active autonomous transaction but do not exit the autonomous routine. When one transaction ends, the next SQL statement begins another transaction. A single autonomous routine can contain several autonomous transactions, if it issues several `COMMIT` statements.

Savepoints

The scope of a savepoint is the transaction in which it is defined. Savepoints defined in the main transaction are unrelated to savepoints defined in its autonomous transactions. In fact, the main transaction and an autonomous transaction can use the same savepoint names.

You can roll back only to savepoints marked in the current transaction. In an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

Avoiding Errors with Autonomous Transactions

You cannot run a `PIPE ROW` statement in your autonomous routine while your autonomous transaction is open. You must close the autonomous transaction before running the `PIPE ROW` statement. This is normally accomplished by committing or rolling back the autonomous transaction before running the `PIPE ROW` statement.

To avoid some common errors, remember:

- If an autonomous transaction attempts to access a resource held by the main transaction, a deadlock can occur. The database raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.
- The database initialization parameter `TRANSACTIONS` specifies the maximum number of concurrent transactions. That number might be exceeded because an autonomous transaction runs concurrently with the main transaction.

- If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception goes unhandled, the transaction is rolled back.

Autonomous Triggers

A trigger must be autonomous to run TCL or DDL statements. To run DDL statements, the trigger must use native dynamic SQL.

See Also:

- Chapter 9, "PL/SQL Triggers," for general information about triggers
- "Description of Static SQL" on page 6-1 for general information about TCL statements
- *Oracle Database SQL Language Reference* for information about DDL statements
- "Native Dynamic SQL" on page 7-2 for information about native dynamic SQL

One use of triggers is to log events transparently—for example, to log all inserts into a table, even those that roll back. In Example 6–47, whenever a row is inserted into the `EMPLOYEES` table, a trigger inserts the same row into a log table. Because the trigger is autonomous, it can commit changes to the log table regardless of whether they are committed to the main table.

Example 6–47 Autonomous Trigger Logs INSERT Statements

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

-- Log table:

DROP TABLE log;
CREATE TABLE log (
  log_id   NUMBER(6),
  up_date  DATE,
  new_sal  NUMBER(8,2),
  old_sal  NUMBER(8,2)
);

-- Autonomous trigger on emp table:

CREATE OR REPLACE TRIGGER log_sal
  BEFORE UPDATE OF salary ON emp FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO log (
    log_id,
    up_date,
    new_sal,
    old_sal
  )
  VALUES (
    :old.employee_id,
    SYSDATE,
```

```
    :new.salary,
    :old.salary
  );
  COMMIT;
END;
/
UPDATE emp
SET salary = salary * 1.05
WHERE employee_id = 115;

COMMIT;

UPDATE emp
SET salary = salary * 1.05
WHERE employee_id = 116;

ROLLBACK;

-- Show that both committed and rolled-back updates
-- add rows to log table

SELECT * FROM log
WHERE log_id = 115 OR log_id = 116;
```

Result:

```
    LOG_ID UP_DATE      NEW_SAL    OLD_SAL
---------- --------- ---------- ----------
       115 28-APR-10    3417.75       3255
       116 28-APR-10    3197.25       3045

2 rows selected.
```

In Example 6–48, an autonomous trigger uses native dynamic SQL (an `EXECUTE IMMEDIATE` statement) to drop a temporary table after a row is inserted into the table `log`.

Example 6–48 Autonomous Trigger Uses Native Dynamic SQL for DDL

```
DROP TABLE temp;
CREATE TABLE temp (
  temp_id NUMBER(6),
  up_date DATE
);

CREATE OR REPLACE TRIGGER drop_temp_table
  AFTER INSERT ON log
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE temp';
  COMMIT;
END;
/
-- Show how trigger works
SELECT * FROM temp;
```

Result:

```
no rows selected
```

```
INSERT INTO log (log_id, up_date, new_sal, old_sal)
VALUES (999, SYSDATE, 5000, 4500);

1 row created.

SELECT * FROM temp;
```

Result:

```
SELECT * FROM temp
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

Invoking Autonomous Functions from SQL

A function invoked from SQL statements must obey rules meant to control side effects (for details, see"Subprogram Side Effects" on page 8-34).

By definition, an autonomous routine never reads or writes database state (that is, it neither queries nor modifies any database table).

The package function `log_msg` in Example 6–49 is autonomous. Therefore, when the query invokes the function, the function inserts a message into database table `debug_output` without violating the rule against writing database state (modifying database tables).

Example 6–49 Invoking Autonomous Function

```
DROP TABLE debug_output;
CREATE TABLE debug_output (message VARCHAR2(200));

CREATE OR REPLACE PACKAGE debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2;
END debugging;
/
CREATE OR REPLACE PACKAGE BODY debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
  BEGIN
    INSERT INTO debug_output (message) VALUES (msg);
    COMMIT;
    RETURN msg;
  END;
END debugging;
/
-- Invoke package function from query
DECLARE
  my_emp_id    NUMBER(6);
  my_last_name VARCHAR2(25);
  my_count     NUMBER;
BEGIN
  my_emp_id := 120;

  SELECT debugging.log_msg(last_name)
  INTO my_last_name
  FROM employees
  WHERE employee_id = my_emp_id;

  /* Even if you roll back in this scope,
     the insert into 'debug_output' remains committed,
```

```
     because it is part of an autonomous transaction. */

  ROLLBACK;
END;
/
```