Being Explicitly Smart with Implicit Cursors

In some cases, Oracle doesn't require you to manually create a cursor as a way of accessing existing data. Instead of *explicit cursors,* it uses *implicit cursors* and drives them automatically so that no more OPEN/FETCH/CLOSE commands are needed. You need to do less coding than the explicit cursors (which we discuss in the rest of this chapter), and implicit cursors sometimes even execute a tiny bit faster than the corresponding explicit cursor.

Although implicit cursors make coding easier in some regards, they can be tricky to work with unless you're careful. In the following section, you find out how to use a basic implicit cursor and how to avoid problems when using them.

Retrieving a single row: The basic syntax

If you're retrieving a single row of data (like information for a single employee, or for a count of employees in a single department), you can use an implicit cursor. You do not even need to specify the cursor. You can use a SELECT INTO command. For example, to get the count of all employees in an organization, you might write something like the following example:

```
declare
    v_out_nr NUMBER;
begin
    select count(*) into v_out_nr                          →4
    from emp;
    DBMS_OUTPUT.put_line
       ('the number of emps is:'||v_out_nr);
end;
```

→4 Takes the place of the explicit cursor declaration as well as opening, fetching, and closing the cursor. All cursor activity is replaced by a single SELECT INTO command.

This code is much easier to write than an explicit declaration of the cursor with an associated OPEN/FETCH/CLOSE code sequence. Behind the scenes, Oracle is still creating a cursor called an implicit cursor.

To use a SELECT INTO command, the query must return exactly one row. If the SELECT statement returns no rows, the code will throw a NO_DATA_FOUND exception. If it returns more than one row, the code will throw the TOO_MANY_ROWS exception. Fortunately, you can still use implicit cursors even if your code might not return any row or more than one row. The next section has the details.

Handling exceptions in implicit cursors

You can still use an implicit cursor with the SELECT INTO command even if there is a possibility of returning no rows (or more than one row) from the query. For example, if you want to have a function that returns the name of a department given its department number, there's a chance someone might enter a nonexistent department number, and you need a way to handle that scenario. In that situation, you might write your function like this:

```
create or replace
function f_getdName_tx (in_deptNo NUMBER)
return VARCHAR2 is
    v_out_tx dept.dName%TYPE;
begin
    select dName into v_out_tx
    from dept
    where deptNo =  in_deptNo;
    return v_out_tx;
exception
    when no_data_found then
        return 'NO SUCH DEPARTMENT';
end f_getdName_tx;
```

In this example, because deptNo is the primary key of the table, you don't have to worry about the query returning too many rows. But if a user asked for the name of a department that doesn't exist, the situation would be addressed in the exception handler.

Returning an implicit cursor into a record

One downside of an implicit cursor is that there is no easy way to declare a record into which to return the cursor. There are two workarounds that you might find useful.

First, if the query columns from the cursor are the same as the columns in a single table, you can use the %ROWTYPE clause on the table name as shown Listing 6-12.

Listing 6-12: Using the %ROWTYPE Clause on the Table Name

```
declare
    r_emp emp%ROWTYPE;                                        →2
begin
    select emp.* into r_emp                                   →4
    from emp,
         dept
    where emp.deptNo = dept.deptNo
```

(continued)

Listing 6-12 ***(continued)***

```
     and   emp.deptNo = 20
     and emp.job = 'MANAGER';
     DBMS_OUTPUT.put_line
         ('Dept 20 Manager is:'||r_emp.eName);
end;
```

Check out the details about lines 2 and 4:

- **→2** Declares a record based on the `EMP` table because the cursor uses the same structure for the records returned by the cursor.
- **→4** Fetches the implicit cursor into the record defined in line 2.

In Listing 6-12, the query returns only the columns from the `EMP` table, so you could specify the cursor record by using the `EMP` table.

Another possible workaround might be needed if the cursor returns many columns from different tables. In this case, you could explicitly declare a record variable, as we discuss in the previous section "Returning more than one piece of information."

Accessing Status Info by Using Cursor Variables

Oracle can tell you the status of a cursor. Specifically, you can find out

- Whether the cursor is open
- Whether a row was found the last time the cursor was accessed
- How many records have been returned

All cursors have properties that report their state of operation. For example, in Listing 6-5, earlier in this chapter, the syntax `%NOTFOUND` is used to terminate a loop. Because the syntax used to capture the state of or information about cursors enables you to make decisions in your code, they are called "cursor variables." There are four variables:

- `%FOUND` checks whether a fetch succeeded in bringing a record into a variable. Returns `TRUE` if the fetch succeeded, `FALSE` otherwise.
- `%NOTFOUND` the reverse of `%FOUND`. Returns `FALSE` if the fetch succeeded, `TRUE` otherwise.
- `%ISOPEN` checks whether a cursor is open.
- `%ROWCOUNT` returns the number of rows processed by a cursor at the time the `%ROWCOUNT` statement is executed.

The variable properties of explicit cursors are referenced as

```
cursor_name%VARIABLE_NAME
```

Specifically, to reference the `c_emp` cursor with the `%FOUND` variable, you'd do it like this:

```
c_emp%FOUND
```

For implicit cursors, the syntax is always *`sql%variable_name`*. The following section explains how to use cursor variables with both explicit and implicit cursors in more detail.

The last variable, `%ROWCOUNT`, is a regular number variable, but the first three are Boolean variables that return a logical `TRUE` or `FALSE`. They can be used together with other logical expressions. For example, if you want to ensure that the fetch succeeded and then check the value returned, you can combine them as follows:

```
if c_empInDept%FOUND and r_emp.eName = 'King'...
```

Checking the status of explicit cursors

The following example illustrates how to use cursor variables with explicit cursors. Listing 6-13 shows the values of cursor variables on a cursor that loops through employee names in a department.

Listing 6-13: Using Explicit Cursors

```
Declare
    cursor c_emp (cin_deptNo NUMBER) is
      select eName
        from emp
        where deptNo=cin_deptNo;
    v_eName VARCHAR2(256);
begin
    if not c_emp%ISOPEN then                              →8
        DBMS_OUTPUT.put_line('Cursor is closed');
    end if;

    open c_emp(10);

    if c_emp%ISOPEN then                                  →14
        DBMS_OUTPUT.put_line('Cursor is opened');
    end if;

    loop
        fetch c_emp into v_eName;
```

(continued)

Listing 6-13 *(continued)*

```
        if c_emp%NOTFOUND then                                →20
            DBMS_OUTPUT.put_line('No rows to fetch!');
            exit; -- the same as exit when c1%NOTFOUND;
        end if;

        DBMS_OUTPUT.put_line
            ('Processed:'||c_emp%rowcount);                   →26
    end loop;

    close c_emp;

    if not c_emp%ISOPEN then                                  →31
        DBMS_OUTPUT.put_line('Cursor is closed');
    end if;
end;
```

In this case, the output of Listing 6-13 would be:

```
Cursor is closed
Cursor is opened
Processed:1
Processed:2
Processed:3
No rows to fetch!
Cursor is closed
```

Using `%ISOPEN` showed exactly when the cursor was opened; `%ROWCOUNT` showed the number of currently fetched rows; and `%NOTFOUND` showed when there were no more rows to fetch.

There are some issues to be aware of:

- If you use the `%FOUND`, `%NOTFOUND`, and `%ROWCOUNT` cursor variables before the cursor is opened or after the cursor is closed, they will raise an exception. If you see an exception from this situation, you probably made a mistake in your code.
- Values of `%FOUND`, `%NOTFOUND`, and `%ROWCOUNT` are changed after every fetch. So, the status of these variables refers to the status of the cursor after the last fetch from the cursor.
- If there are no more rows to fetch, `%ROWCOUNT` keeps the number of successfully fetched records until the cursor is closed. No matter how many unsuccessful fetches you make from a cursor, the value of this variable won't change.

Checking the status of implicit cursors

You can use the same cursor variables for implicit cursors, too. When used with an implicit cursor, the value of a cursor variable corresponds to the last statement needing an implicit cursor that was fired in the current procedural block (the area between `BEGIN` and `END`). Because there is no cursor name, you use `SQL` rather than the cursor name. In the following example, look at the value of cursor variables on an implicit cursor that updates an employee's salary:

```
SQL> begin
  2      update emp
  3      set sal=sal*1
  4      where eName='KING';
  5
  6      DBMS_OUTPUT.put_line('Processed:'||sql%rowcount);
  7
  8      if sql%FOUND then
  9          DBMS_OUTPUT.put_line('Found=true');
 10      else
 11          DBMS_OUTPUT.put_line('Found=false');
 12      end if;
 13  end;
 14  /
Processed:1
Found=true
PL/SQL procedure successfully completed.
SQL>
```

As you can see from this example, cursor variables are wonderful tools for knowing exactly how many records were processed and whether any were processed at all.

In the preceding example, if you change the `WHERE` clause to `where eName='TEST';` the output changes, as shown here:

```
SQL> begin
  2      update emp
  3      set sal=sal*1
  4      where eName='TEST';
  5
  6      DBMS_OUTPUT.put_line('Processed:'||sql%rowcount);
  7
  8      if sql%FOUND then
  9          DBMS_OUTPUT.put_line('Found=true');
 10      else
 11          DBMS_OUTPUT.put_line('Found=false');
 12      end if;
 13  end;
 14  /
Processed:0
Found=false
PL/SQL procedure successfully completed.
SQL>
```

Because there is no employee with the specified name, no row was updated. You don't need to requery the table to find out how many records were updated. This last point is especially critical for batch processing. By taking advantage of SQL%ROWCOUNT, you can reduce the amount of code you have to write to detect whether your DML operations were successful.

WARNING!

As usual, there are some caveats in using implicit cursor variables:

- %ISOPEN is always false because implicit cursors are opened as needed and closed immediately after the statement is finished. Never use %ISOPEN with implicit cursors.
- Both %FOUND and %NOTFOUND are false before any statement is executed. Using them in your code in this way is a waste of time and space.
- Any DDL or transaction control commands (commit or rollback) will clear implicit cursor variables. You can check the value of your cursor variables only prior to doing a commit or rollback.

Updating Records Fetched from Cursors

As you loop through a set of records, you'll frequently want to update each of the records. For example, you might want to loop through all employees and adjust their salaries or loop through purchase orders and update their statuses.

You can update records in a few different ways. When you need to use more complex logic, be sure to evaluate whether to lock the records so that other actions don't interfere with your update. The following sections explain the different methods and considerations in more detail.

Using a simple UPDATE statement

Often, you can update records with a simple SQL UPDATE statement like

```
update emp
set salary = salary * 1.1
where deptNo = 10;
```

This statement would add 10 percent to everyone's salary in department 10.

However, sometimes you need to use more complex logic.

Updating with logical operators

When you need to look at each record individually and decide what to do with it, logical operators come into play. You usually don't want to simply update all the records the same way. One way to do this is by embedding a simple UPDATE statement in your code. For example, the following code gives a raise to everyone with a salary below $5,000:

```
declare
  cursor c_empInDept is
    select * from emp;
  begin
  for r_emp in c_empInDept loop
    if r_emp.sal < 5000 then
      update emp
      set sal = sal * 1.1
      where empNo = r_emp.empNo;
    end if;
  end loop;
end;
```

But this isn't very safe code. While your routine is running, someone else might be updating employee salaries. You might give a raise to someone whose salary, in the last split second, has already been raised above $5,000.

To solve this problem, you need to lock all the records while you're working on them. This is done using a SELECT FOR UPDATE command, as shown in Listing 6-14. You can find out more about locking, sessions, and transaction control in Chapter 12.

Listing 6-14: Using the SELECT FOR UPDATE Command

```
declare
  cursor c_empInDept is
    select * from emp
    for update of sal;                                  →4
begin
  for r_emp in c_empInDept loop
    if r_emp.sal < 5000 then
      update emp
      set sal = sal * 1.1
      where current of c_empInDept;                    →10
    end if;
  end loop;
end;
```

Here are the details about lines 4 and 10:

→4 Notice that the code uses FOR UPDATE OF SAL;. This lock means that others can't delete records or modify the salary column but are allowed to modify other columns. You need to lock not only the column that you're modifying, but also any other column that might determine what record you will process. If you don't specify any column, the clause FOR UPDATE locks the entire record.

→10 Notice that the WHERE clause was changed to use `where current of c_empInDept`. This code updates the exact record that the cursor is referencing by using the internal Oracle record identifier (`rowid`). It will execute very fast.

If you lock the records, no one else will be allowed to modify the records until your cursor closes. This can be a problem. If your routine takes a long time to execute, you can affect other users of your system.

Whether to use SELECT FOR UPDATE or just UPDATE for the current record by using the primary key is a difficult decision to make. You need to balance the safety of SELECT FOR UPDATE against the impact that it might have on other parts of the system.

Taking a Shortcut with CURSOR FOR Loops

The technique of looping through all the records in a cursor is so common that PL/SQL has a nifty shortcut, the CURSOR FOR loop. This is an alternative to the OPEN/FETCH/CLOSE sequence introduced in "Looping through multiple records," earlier in this chapter.

Of course, if there's a shortcut, you might wonder why anyone still uses OPEN/FETCH/CLOSE at all. And the reason is an important one: There are some exception-handling issues to consider before choosing to use a CURSOR FOR loop implementation. If something goes wrong inside the CURSOR FOR loop, Oracle closes the cursor, which can affect your procedural logic.

The following sections introduce how this shortcut works and help you decide when to take the shortcut and when the long way is better.

Comparing CURSOR FOR loops to cursors with the LOOP command

As we explain in the "Looping through multiple records" section earlier in this chapter, cursors help process multiple records at once in a block of code. For example, if you need to access all the records in department 10, increase the salary of all employees in department 10 by 50 percent, and print out a report with the old and new salary values, the code would look something like Listing 6-15.

Listing 6-15: Looping through a Cursor by Using the LOOP Command

```
declare
    cursor c_emp (ci_deptNo NUMBER) is
      select *
        from emp
        where deptNo = ci_deptNo;
    r_emp c_emp%ROWTYPE;
begin
    open c_emp(10);
    loop
        fetch c_emp into r_emp;
        exit when c_emp%NOTFOUND;
        update emp
        set sal=sal*1.5
        where empNo=r_emp.empNo;
        DBMS_OUTPUT.put_line('Emp '||r_emp.eName||
         ' - salary change:'||r_emp.sal||
         '->'||r_emp.sal*1.5);
    end loop;
    close c_emp;
end;
```

TIP

Although Listing 6-15 will work, if you want to process all the rows in the query, you don't need to bother with the full `OPEN/FETCH/EXIT/CLOSE` syntax. You can make your code more compact by telling Oracle to manage a `CURSOR FOR` loop based on the cursor, as in Listing 6-16.

Listing 6-16: Looping through a Cursor by Using a CURSOR FOR Loop

```
declare
    cursor c_emp (ci_deptNo NUMBER) is
      select *
        from emp
        where deptNo = ci_deptNo;
begin
    for r_emp in c_emp(10) loop
        update emp
        set sal=sal*1.5
        where empNo = r_emp.empNo;
        DBMS_OUTPUT.put_line('Emp '|| r_emp.eName||
         ' - salary change:'||r_emp.sal||'-
         '- >'||r_emp.sal*1.5);
    end loop;
end;
```

The method shown in Listing 6-16 is much more convenient than Listing 6-15. The code is much shorter. Several tasks you previously needed to do by hand are handled automatically:

- You don't have to declare the r_emp record variable. In a CURSOR FOR loop, the record variable is automatically declared when it is used in the FOR statement, and it automatically has the same row type as the cursor.
- There is no EXIT statement. The program automatically exits the loop when there are no more records to process.
- There are no OPEN or CLOSE statements. The CURSOR FOR loop automatically opens the cursor and closes it when there are no more records to fetch.
- If a SQL query returns no records, the code inside the loop won't be executed at all.
- The record that holds data retrieved by the cursor exists only inside the loop and provides temporary storage of each fetched record. Individual columns can be referenced as variable.column_name.

Listing 6-16 is much easier to read and understand, but it will spin through the dataset and process all its records just the same as the longer version.

When do CURSOR FOR loops simplify exception handling?

By using CURSOR FOR loops, you don't need to worry about an accumulation of open cursors that could eventually reach the maximum number of cursors allowed for the database. For example, if you were looping through employees doing payroll and encountered an error, you would have to stop processing your payroll and gracefully recover. The following example illustrates the simplification of code made possible by the CURSOR FOR loop's automatic housekeeping:

```
declare
    cursor c_emp is ...
begin
    for r_emp in c_emp loop
      <...something that could fail ...>
    end loop;
exception
    when others then
         /* The cursor is already closed - don't do
          anything*/
        ...
end;
```

If you write the same example with OPEN/FETCH/CLOSE, you need to know whether failure can occur before the cursor is closed.

```
declare
    cursor c_emp is ...
    r_emp c_emp%ROWTYPE;
begin
    open c1;
    loop
        fetch c_emp into r_emp;
        exit when c_emp%NOTFOUND;
        <... .something that could fail #1..>
    end loop;
    close c_emp;
        <... .something that could fail #2..>
exception
    when others then
         /* Cursor is opened at #1 and closed at #2 */
         if c_emp%ISOPEN is true
         then
            close c_emp; -- close the cursor
         end if;
        raise;
end;
```

If failure occurs during loop processing (in the first example), you must close the cursor in the exception block. But you need to check %ISOPEN beforehand, because the failure could have occurred in processing the second example code block, and the cursor would already be closed.

Make it a habit to place such exception-handling logic into any code that uses explicit cursors (not CURSOR FOR loops). Otherwise, you risk intermittent and unpredictable database collapse.

When should you use a CURSOR FOR loop instead of a looping through the cursor by using a LOOP command? The answer is pretty simple: Always start by using a CURSOR FOR loop. If you then end up needing very precise control that prevents you from using a CURSOR FOR loop, change your code to the traditional technique. Only rarely will you need to switch to the traditional way.

When CURSOR FOR loops make your life harder

There are times when you want to use the standard OPEN/FETCH/CLOSE syntax in a loop. In a CURSOR FOR loop, if you want to know the last record retrieved, you have to do some extra work. In the basic OPEN/FETCH/CLOSE sequence, the last fetched value before failure is sitting in the record variable, but that isn't the case in a CURSOR FOR loop. In a CURSOR FOR loop, the record variable is null after the cursor closes.

TECHNICAL STUFF

A shortcut in a shortcut

If you want to push the envelope to the extreme, you can rewrite Listing 6-13 without explicitly declaring a `CURSOR FOR` loop at all. Although we don't recommend this strategy, because of the decreased readability of the code and lack of reusability of a cursor, you can still do it, as shown here:

```
begin
    for r_emp in (select *
                from emp
                where deptNo = 10) loop
        <... do something ...>
    end loop;
end;
```

To solve this problem, you could store part or all of the current record in a variable visible outside the `CURSOR FOR` loop, and then you could perform whatever processing you need. If a failure occurs, you can use that record in the exception handler, as shown in Listing 6-17.

Listing 6-17: Adding an Exception Handler to a CURSOR FOR Loop

```
declare
    cursor c_emp (ci_deptNo NUMBER) is
      select empNo, deptNo, empName
        from emp
        where deptNo = ci_deptNo
        order by empNo; --helps ID failure point        →6
    v_empNo NUMBER;
begin
    for r_emp in c_emp(10) loop
        v_empNo := r_emp.empNo; --record identifier   →10
         <... .something that could fail #1..>
    end loop;
exception
    when others then
      raise_application_error
        (-20999,'Update failed on the emp#'||v_empNo||
         ' with error :'||sqlerrm);                    →17
end;
```

Here's how Listing 6-17 works:

→10 Saves `EMPNO` into a variable (`v_empNo`) that will still exist after the cursor is closed.

→17 References `v_empNo` in the error handler. You know that all records with `empNo < V_empNo` were processed successfully because of the `order by` (line 6) in the query.

Knowing what record is processing

Cursor variables were introduced earlier in the section "Accessing Status Info Using Cursor Variables." Although the `%ISOPEN`, `%FOUND`, `%NOTFOUND` variables aren't useful at all in `CURSOR FOR` loops, you can use `%ROWCOUNT` to detect what record you are processing at a given point, as shown in the following example:

```
declare
    v_recordCount_nr NUMBER;                              →2
    cursor c1 is ...
begin
    for r in c1 loop
        v_recordCount_nr:=c1%rowcount;                    →6
        <... do something ...>
    end loop;
    DBMS_OUTPUT.put_line('Rows processed:'||
          v_recordCount_nr);                             →10
end;
```

Here are some additional details about this code:

→2 Declares a variable that will hold the number of records processed.

→6 Copies the number of records fetched into the variable.

→10 References the number of records retrieved after the cursor is closed.

Because you need to know the value of the cursor variable outside the loop, you have to use an additional variable to store that value. If you don't do this, after the loop is closed you can't answer the question "How many records were processed?" Because the cursor is already closed, there is no way to access its information. As usual, if you need something done, just do it yourself.

Referencing Functions in SQL

In SQL, you already know that you can retrieve columns as well as expressions that reference those columns. What you may not know is that you can also write your own functions that can then be referenced within the query.

You find out about writing functions in Chapter 4. If you haven't read that chapter yet, you might want to read it now.

Why some cursor variables work, and some don't

%ISOPEN, %FOUND, %NOTFOUND variables aren't useful at all in CURSOR FOR loops:

- The CURSOR FOR loop is always closed before and after the loop and open inside the loop. There is no reason to ever use %ISOPEN.
- Inside the loop, records are always found (or else the program would never go inside the loop). So %FOUND is always true inside the loop. Outside the loop, %FOUND, %NOTFOUND would return an error.

%ROWCOUNT is useful:

- %ROWCOUNT can be referenced inside the loop in CURSOR FOR loops or if you used an explicit cursor to define the loop. You can't use it with implicit cursors.
- %ROWCOUNT can't be used before or after the loop. It will return an error.

Start with a simple example of creating a Departments/Employees report sorted by department, with a comma-separated list of employees in each department. Your report should look like this:

Department	***Employees***
Accounting	Smith, Jones
Finance	Benson, Marks, Carson
Marketing	Johnson, Chu

There are actually two tasks required here: preparing the comma-separated list and displaying the report.

1. **To prepare the comma-separated list, create the following PL/SQL function, named `f_get_Emps`:**

```
create or replace function f_getEmps_tx
   (i_deptNo NUMBER)
return VARCHAR2
is
    cursor c_emp is
      select eName
        from   emp
        where  deptNo = i_deptNo;
    v_out_tx VARCHAR2(4000);
begin
    for r_emp in c_emp loop
        if v_out_tx is null
        then
            v_out_tx:=r_emp.eName;
        elsif length(v_out_tx)+
              length(r_emp.eName)>3999
        then
```

```
            null;
        else
            v_out_tx:=v_out_tx||', '||r_emp.eName;
        end if;
    end loop;
    return v out tx:
end;
```

2. **Display the report:**

```
select deptNo, dname, f_getEmps_tx(deptNo) emps
from dept
order by dname;
```

That's all you need to do. You can use PL/SQL functions inside SQL code, which gives you the power to go beyond pure SQL functionality. You can use "industrial strength" procedural logic to work with requirements that you couldn't implement otherwise.

Important facts to remember

Everything always comes with a price, and using functions in SQL is no exception. There are a number of drawbacks and restrictions to using these functions.

Datatypes

PL/SQL datatypes don't always correspond directly to SQL datatypes (more details about PL/SQL datatypes can be found in Chapters 10 and 11):

- BOOLEAN and REF CURSOR types do not exist in SQL at all.
- VARCHAR2 can only go up to 4,000 in SQL rather than 32,000 in PL/SQL (note this limit in the preceding example).

Read/write restrictions

The PL/SQL reference manual includes some fairly strict rules:

- When called from a SELECT statement, the function cannot modify any data (no DML except SELECT).
- When called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any data.
- When called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). Also, it cannot execute DDL statements (such as CREATE) because they are followed by an implicit COMMIT.

The reason for these rules is simple. Oracle can't be sure that modifying data, the session, the system, or object structure doesn't have any impact on the data you're querying or even on objects you're processing. If such activity isn't blocked, a logical loop or conflict that can't be resolved might result.

Think about what should happen if the function in the next example is called in SQL. This function updates the salary of the specified employee and tells you whether the update was successful:

```
create or replace
function f_giveRaise_tx (i_empNo NUMBER, i_pcnt NUMBER)
return VARCHAR2 is
begin
    update emp
      set sal=sal*(i_pcnt/100)+sal
      where empNo = i_empNo;
    return 'OK';
exception
    when others then
        return 'Error:'||substr(sqlerrm,1,256);
end f_giveRaise_tx;
```

Instead of the update confirmation, the result of the query is an Oracle error, which is caught by the exception handler of the function:

```
SQL> select f_giveRaise_tx(7369,100)
  2  from dual;

F_GIVERAISE_TX(7369,100)
----------------------------------------------------------
Error:ORA-14551: cannot perform a DML operation inside a
          query
SQL>
```

Oops! Oracle just told you that you cannot make that `UPDATE`.

Performance impact

How does the Oracle know what exactly is happening "under the hood" of the function that you placed inside the query? How can it determine what impact that function could have on overall execution? It can't.

In terms of performance, using functions in SQL is risky. With functions in SQL, the whole idea of SQL optimization gains another dimension; namely, decreasing the impact of function calls.

There are some guidelines you can follow. The next example shows a display function for an employee that returns name and job. It also includes a view that uses this display function for managers:

```
create or replace
function f_emp_dsp (i_empNo NUMBER)
return VARCHAR2 is
    v_out_tx VARCHAR2 (256);
begin
    DBMS_OUTPUT.put_line('Inside of F_EMP_DSP');
    select initcap(eName)||': '||initcap(job)
      into v_out_tx
      from emp
      where empNo = i_empNo;
    return v_out_tx;
end f_emp_dsp;
/
create or replace view v_emp as
select empNo, eName, mgr, f_emp_dsp(mgr) mgr_name, deptNo
from emp;
/
```

When you query the view, it may run much more slowly than a query that accesses the EMP table directly. If performance is important (and performance is always important), you need to be careful. Here are some guidelines:

Don't ask for what you don't need

If you only need EMPNO and ENAME, the following statement is inefficient:

```
select *
from v_emp;
```

Use this statement instead:

```
select empNo, eName
from v_emp;
```

REMEMBER

Remember that one of the columns in the view v_emp is defined as a function. In the first case, that function will be executed for each record to be processed. The asterisk (*) means that you are retrieving all columns listed in the view including the column defined as a function. You do not need that extra data, but it will still be unnecessarily calculated when the query is executed, making your query run more slowly than necessary.

Don't ask for what you already have

Function f_emp_dsp will return exactly the same value for the same employee each time it is called. This behavior is called "deterministic." Knowing about this behavior can help Oracle avoid redundant function calls. If a deterministic function was called previously with the same arguments, the optimizer can elect to use the previous result. Thus, the function could be modified as follows:

```
create or replace function f_emp_dsp (in_empNo NUMBER)
return VARCHAR2
DETERMINISTIC is
...
```

Declaring a function `DETERMINISTIC` is only a hint, and there is no guarantee that the optimizer will use it. However, it can be very handy.

Don't run the function all the time when you only need it some of the time

Assume that you need to take some action for every record in department 10, which includes using the display function for employees. You could start by writing your query this way:

```
declare
    cursor c_emp is
      select *
        from v_emp;
begin
    for r_emp in c_emp loop
        if r_emp.deptNo = 10 then
            ...
        end if;
    end loop;
end;
```

You should assume that any number of calls greater than the number of employees in department 10 is a waste of resources. The following query works exactly as expected:

```
declare
    cursor c_emp is
      select *
        from v_emp
        where deptNo=10;
begin
    for r_emp in c_emp loop
    ...
    end if;
end;
```

Function calls can be expensive from a system resource perspective. Do your best to ensure that the calls you use are efficient and do only what you want them to do.

Getting good performance with functions

Oracle can't do everything for you. For example, it can't guess exactly what you want from your system. The human mind can always outsmart a computer, but the trick is not to outsmart yourself.

Sticking to the following rules will make your life and your database performance significantly better.

- **As you write any function, ask yourself, "Will it be used in SQL or not?"** If so, verify that SQL works with the datatypes you're passing in and out.
- **Verify that you are not performing illegal reads/writes.** For how to cheat if needed, see Chapter 12, which covers transaction control.
- **Think about performance at design time, not later, when users start to complain about it.** Write your code with its future use in mind. Sometimes saving a keystroke or two in implementation might seem like a good idea, but it can result in hours of necessary tuning when your system is in production.