Working with Oracle SQL

Chapter 9:
Programming with PL/SQL

Chapter Objectives

In this chapter, we will discuss:

- Overview of PL/SQL
- Declaring, assigning, and using scalar variables
- Defining and using conditional, iterative, and sequential control
- Utilizing error handling and writing exception handlers
- Processing result sets with cursors
- Improving cursor processing with `FOR-LOOP` cursors
- Improving update and delete performance by using cursors

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

PL/SQL Example

- Requirement:
  - Employee William Smith (171) wishes to become an IT Programmer
  - In our organization, we are limited to a maximum of 5 programmers
    - If present number is less than the maximum, promote William
    - Otherwise, add him to a waiting list for that position

```
DECLARE
    no_of_employees          NUMBER(2);
    max_employees   CONSTANT NUMBER(1) := 5;
BEGIN
    SELECT COUNT(*)
    INTO no_of_employees
    FROM employees
    WHERE job_id = 'IT_PROG';

    IF no_of_employees < max_employees THEN
        UPDATE employees
        SET job_id = 'IT_PROG'
        WHERE employee_id = 171;
    ELSE
        INSERT INTO waiting_list (
            employee_id,
            job_id
        )
        VALUES (171, 'IT_PROG');
    END IF;
END;
/
```

PL/SQL Basics

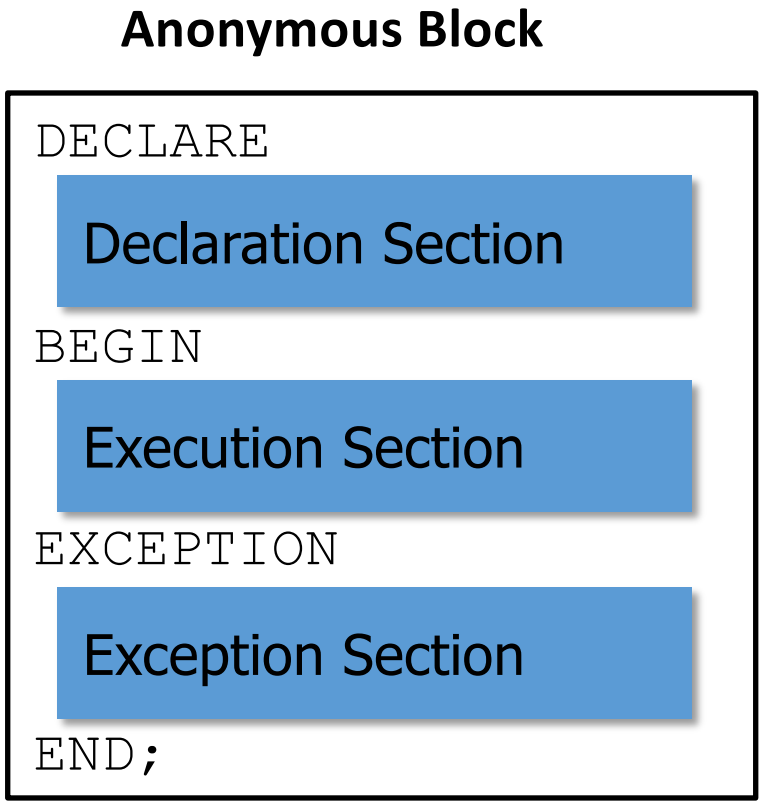
- PL/SQL stands for Procedural Language (PL) extensions to Structured Query Language (SQL)
- Developed by Oracle to be their standard programming language
- Each PL/SQL statement must end with a semicolon
  - Because an `IF` statement is not complete until the `END IF`, a semicolon is required after `END IF` but not allowed after `THEN`
- Similarly, because a block is not complete until the `END` key word, a semicolon is required after `END` but not allowed after `DECLARE` or `BEGIN`
- In PL/SQL, a semicolon does not execute the block (unlike in SQL)
  - A slash (`/`) is required to execute the block
  - The rest of the course notes will not include a slash

PL/SQL Basics (continued)

- The code can be stored in the database
  - Procedures and functions
  - Packages
  - Triggers
  - Can pass parameters
- Provides exception handling
  - Simplifies coding of checking for errors and their resolution
- A PL/SQL program uses embedded SQL statements to access the database tables
  - Similar to SQL statements embedded in other procedural languages

The Structure of PL/SQL

- Each PL/SQL block is executed as a statement
  - `DECLARE … BEGIN … EXCEPTION … END`
    - With other statements embedded
- The `DECLARATION` section contains declarations
  - Variables, constants, exceptions, cursors, etc.
  - The declaration section is optional
- The `EXECUTION` section contains executable statements
  - Each block must have at least one executable statement
    - A mandatory section
- The `EXCEPTION` section contains executable statements for handling errors
  - This section is optional



Variables

- Variables are declared in the declaration section

```
var_name [CONSTANT] datatype | table_name.column_name%TYPE [:= expression];
```

- `%TYPE`
  - "Anchored declaration" provides the datatype of a variable based on a database column
  - Don't need to change code if column definition changes

    ```
    salary employees.salary%TYPE;
    ```

    `salary` is `NUMBER(8, 2)` based on the `employees` column definition
- Assignment operator (`:=`) is used to assign values to variables
  - Initialize in the declaration section

    ```
    max_students NUMBER(2) := 5;
    ```
  - Assign in the executable section

    ```
    salary := salary * 1.10;
    ```
- Use the optional `CONSTANT` keyword if the value of the variable should never be changed

```
max_employees CONSTANT NUMBER(1) := 5;
```

DBMS_OUTPUT Package

- Display variables using DBMS_OUTPUT package
  - Example:

```
DBMS_OUTPUT.PUT_LINE ('salary: ' || salary);
```

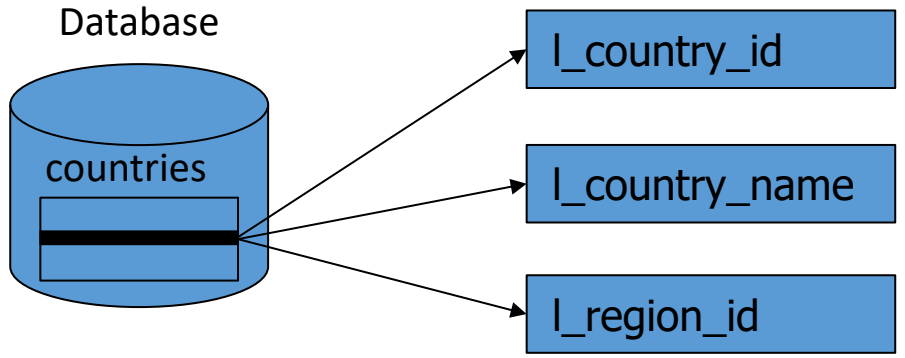
  - Use SET SERVEROUTPUT ON command before executing the procedure
- Important procedures
  - PUT_LINE prints a whole line, including an end of line
  - PUT prints a partial line
  - NEW_LINE prints an end of line and may be used to end a line started with PUT

Working with Variables

- Example: declare variables based on the `countries` table columns

```
DECLARE
    l_country_id   countries.country_id%TYPE;
    l_country_name countries.country_name%TYPE;
    l_region_id    countries.region_id%TYPE;
BEGIN
    SELECT country_id, country_name, region_id
    INTO   l_country_id, l_country_name, l_region_id
    FROM   countries
    WHERE  region_id = 2
    AND    ROWNUM <= 1;

    -- Do something with the data
END;
```



- `ROWNUM` indicates the order of selected rows
  - Can be used to restrict the `SELECT INTO` to return one row
    - `SELECT... INTO` must have exactly one row (0 or more than 1 will cause an error)

Table Record Variables

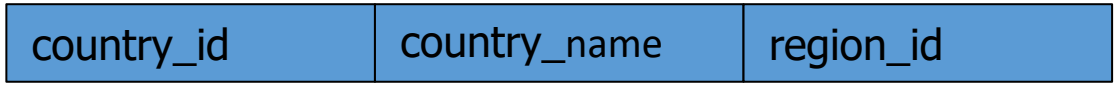
- Combines related variables so that they can be manipulated as a unit

```
record_name     table_name%ROWTYPE;
```

- Example: declare a record based on the countries table row

```
country_rec     countries%ROWTYPE;
```

  - country_rec:



- Select a row into the record type variable

```
SELECT *
INTO   country_rec
FROM   countries
WHERE  region_id = 2
AND    ROWNUM <= 1;
```

- Add a new country to the region

```
country_rec.country_id := 'PE';
country_rec.country_name := 'PERU';

INSERT INTO countries
VALUES country_rec;
```

Naming Variables

- Variables are often named with a prefix or suffix:
  - `c_country_id` (a type-based prefix, indicates it is character data)
  - `l_country_id` (a scope-based prefix, indicates it is a local variable)
  - Avoids clashes with column names, consider this (pointless) code:

```
DECLARE
    country_id   countries.country_id%TYPE := 'BR';
    country_name countries.country_name%TYPE;
    region_id    countries.region_id%TYPE;
BEGIN
    SELECT country_id, country_name, region_id
    INTO   country_id, country_name, region_id
    FROM   countries
    WHERE  region_id = 2
    AND    country_id = country_id
    AND    ROWNUM <= 1;

    DBMS_OUTPUT.PUT_LINE(country_id);
END;
```

No confusion here because only variables can be the target of `SELECT… INTO`

One of these is meant to be the variable, but the column name is favored

Prints AR, not BR

Naming Variables (continued)

- Variables can always be qualified by the name of the enclosing block
  - More flexible and readable than a naming convention
  - Safer, in the (unlikely) event that a column called `l_country_id` is added to the table
- But, aren't these anonymous blocks?!
- Even anonymous blocks can have a (temporary) block name
  - Not stored, discarded after execution
- This course contains both styles

```
<<country_check>>
DECLARE
    country_id   countries.country_id%TYPE := 'BR';
    country_name countries.country_name%TYPE;
    region_id    countries.region_id%TYPE;
BEGIN
    SELECT c.country_id, c.country_name, c.region_id
    INTO   country_check.country_id,
           country_check.country_name,
           country_check.region_id
    FROM   countries c
    WHERE  c.region_id  = 2
    AND    c.country_id = country_check.country_id
    AND    ROWNUM <= 1;

    DBMS_OUTPUT.PUT_LINE(country_check.country_id);
END;
```

Block name

No ambiguity

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

Conditional Control

- `IF` statement
  - Each `IF` clause contains one or more PL/SQL statements
- `CASE` statement
  - Each `WHEN` clause contains one or more PL/SQL statements
  - Supports one or more conditions or one selector
- `CASE` expression
  - Each `WHEN` clause is a single expression
  - Supports one or more conditions or one selector

IF Statement

- Increase salary based on employees' commission

```
DECLARE
    n_employee_id         employees.employee_id%TYPE := 170;
    n_commission          employees.commission_pct%TYPE;
    n_allowance           NUMBER(4);
BEGIN
    SELECT NVL(commission_pct,0)
    INTO n_commission
    FROM employees
    WHERE employee_id = n_employee_id;

    IF TRUNC(n_commission * 100) = 0 THEN
        n_allowance := 500;
    ELSIF TRUNC(n_commission * 100) = 10 THEN
        n_allowance := 400;
    ELSIF TRUNC(n_commission * 100) = 20 THEN
        n_allowance := 300;
    ELSE
        n_allowance := 200;
    END IF;

    UPDATE employees
    SET salary = salary + n_allowance
    WHERE employee_id = n_employee_id;
END;
```

CASE Statements

- Using conditions

```
CASE
    WHEN TRUNC(n_commission * 100) = 0 THEN
        n_allowance := 500;
    WHEN TRUNC(n_commission * 100) = 10 THEN
        n_allowance := 400;
    WHEN TRUNC(n_commission * 100) = 20 THEN
        n_allowance := 300;
    ELSE
        n_allowance := 200;
END CASE;
```

- Using a selector
  - More readable
  - Only suitable for equality conditions

```
CASE TRUNC(n_commission * 100)
    WHEN 0 THEN
        n_allowance := 500;
    WHEN 10 THEN
        n_allowance := 400;
    WHEN 20 THEN
        n_allowance := 300;
    ELSE
        n_allowance := 200;
END CASE;
```

CASE Expressions

- Using conditions

```
n_allowance :=
    CASE
        WHEN TRUNC(n_commission * 100) =  0 THEN 500
        WHEN TRUNC(n_commission * 100) = 10 THEN 400
        WHEN TRUNC(n_commission * 100) = 20 THEN 300
        ELSE 200
    END;
```

- Using a selector
  - Most readable
  - Again, only suitable for equality conditions

```
n_allowance :=
    CASE TRUNC(n_commission * 100)
        WHEN 0  THEN 500
        WHEN 10 THEN 400
        WHEN 20 THEN 300
        ELSE 200
    END;
```

Iterative Control

- Allows a sequence of statements to be performed many times
  - Implemented using a `LOOP` statement
- Three forms of a `LOOP` statement
  - `LOOP`
  - `WHILE-LOOP`
  - `FOR-LOOP`

LOOP Statement

- Syntax:

```
LOOP
    sequence of statements
END LOOP;
```

- Requires an EXIT statement with an optional condition

```
EXIT [ WHEN condition ];
```

- Example: iterate until variable i is greater than 100

```
DECLARE
    i NUMBER(3) := 0;
BEGIN
    LOOP
        i := i + 10;
        IF i > 100 THEN
            EXIT;
        END IF;
    END LOOP;
END;
```

```
DECLARE
    i NUMBER(3) := 0;
BEGIN
    LOOP
        i := i + 10;
        EXIT WHEN i > 100;
    END LOOP;
END;
```

WHILE-LOOP Statement

- Syntax:

```
WHILE condition LOOP
    sequence of statements;
END LOOP;
```

- Evaluates a condition prior to each iteration of the loop
  - The loop is not executed if the condition is initially FALSE or NULL
- Example: iterates until the variable `i` is greater than 100

```
DECLARE
    i NUMBER(3) := 0;
BEGIN
    WHILE i <= 100 LOOP
        i := i + 10;
    END LOOP;
END;
```

FOR-LOOP Statement

- Syntax:

```
FOR counter IN [REVERSE] lower_bound .. upper_bound LOOP
    -- sequence of statements;
END LOOP;
```

- Counter is implicitly declared and value cannot be assigned to it
  - It is of type PLS_INTEGER
  - 32-bit integer, more efficient than NUMBER, only available in PL/SQL
- lower_bound and upper_bound can be variables or constants
  - They are always in the order lower_bound .. upper_bound
  - With the REVERSE key word, the counter is initialized to the upper_bound

FOR-LOOP Example

- Insert all Cartesian products of single-digit integers into a table
  - Use a two-level nested FOR-LOOP

```
BEGIN
    FOR outer_counter IN 1 .. 9 LOOP
        FOR inner_counter IN 1 .. 9 LOOP
            INSERT INTO cartesian_product
            VALUES (outer_counter, inner_counter);
        END LOOP;
    END LOOP;
END;
```

- This example illustrates the scope rules for counter variables
  - outer_counter variable can be referenced in the inner loop
  - inner_counter variable is not available in the outer loop

Exceptions

- Exception section of a block used to process errors
  - Contains one or more exception handlers
  - `OTHERS` key word handles all exceptions that are not explicitly named
    - Must be the last exception handler
- Common predefined exceptions
  - `NO_DATA_FOUND`
    - Raised when a `SELECT INTO` returns no rows
  - `TOO_MANY_ROWS`
    - Raised when a `SELECT INTO` returns more than one row
  - `DUP_VAL_ON_INDEX`
    - Raised when an inserted or updated record violates a unique index
- To stop execution, roll back database changes, and display a message:

  `RAISE_APPLICATION_ERROR(error_number, error_message);`
  - `error_number` is a negative integer between –20000 and –20999
  - `error_message` is a character string up to 2048 bytes in length

Exceptions Example

- Mr. Smith has been promoted to manager of department 150
  - SQLERRM returns Oracle's error message based on the internal error raised

```
DECLARE
    n_employee_id employees.employee_id%TYPE;
BEGIN
    SELECT e.employee_id
    INTO   n_employee_id
    FROM   employees e
    WHERE  e.last_name = 'SMITH';

    UPDATE departments
    SET manager_id = n_employee_id
    WHERE department_id = 150;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20999, 'Employee does not exist');
    WHEN TOO_MANY_ROWS THEN
        RAISE_APPLICATION_ERROR(-20999, 'Multiple employees found');
    WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR(-20999, SQLERRM);
END;
```

WHEN OTHERS

- Be careful with `WHEN OTHERS`
  - Using `RAISE_APPLICATION_ERROR` may mask an unexpected exception
  - Ignoring the exception is worse!
  - Consider doing any processing and then using `RAISE` to re-raise original exception
- Use it sparingly
  - Capturing families of exceptions is hard since Oracle does not have an exception hierarchy
- Do not use it as part of normal processing

Far too many reasons why this may be caught, use `NO_DATA_FOUND` instead

```
<<find_country>>
DECLARE
    country_id   countries.country_id%TYPE := 'XX';
    country_name countries.country_name%TYPE;
BEGIN
    SELECT c.country_name
    INTO   find_country.country_name
    FROM   countries c
    WHERE  c.country_id = find_country.country_id;

    DBMS_OUTPUT.PUT_LINE('Country found with name ' ||
                find_country.country_name);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Country not found');
END;
```

Exercise 9.1: Building Anonymous Blocks

30 min

- Please complete this exercise in your Exercise Manual

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

Implicit Cursors

- Oracle automatically opens a cursor to process each SQL statement
  - Refers only to the last SQL statement executed
- The most recent implicit cursor is referred to as the “SQL” cursor
  - Contains attributes that provide information about the execution of `INSERT`, `UPDATE`, `DELETE`, or `SELECT INTO` statements
- Common attributes:
  - `SQL%FOUND` returns a Boolean value
    - `TRUE` if an `INSERT`, `UPDATE`, or `DELETE` affected one or more rows or a `SELECT INTO` returned one or more rows
  - `SQL%NOTFOUND` is the logical opposite of `SQL%FOUND`
  - `SQL%ROWCOUNT`
    - Returns the number of rows affected by an `INSERT`, `UPDATE`, or `DELETE`, or returned by a `SELECT INTO`
- Attributes can be used in procedural statements but not in SQL statements

Explicit Cursor Loop

- An Explicit Cursor is defined as a `SELECT` statement
- `OPEN` executes the query
  - Sets up the cursor in memory but does not fetch the results
- A `LOOP` is used to retrieve multiple rows from a cursor
  - `FETCH` retrieves a row from the cursor
    - Places it into a cursor record, and advances the cursor to the next row
  - `EXIT` terminates the loop
    - Usually based on a cursor attribute
- `CLOSE` deallocates memory

```
DECLARE
    CURSOR cursor_name
    IS
        select_statement;

    cursor_record cursor_name%ROWTYPE;
BEGIN
    OPEN cursor_name;

    LOOP
        FETCH cursor_name INTO cursor_record;
        EXIT WHEN cursor_name%NOTFOUND;

     ... use the cursor record ...

    END LOOP;

    CLOSE cursor_name;
END;
```

Cursor Record and Cursor Attributes

- Cursor record
  - Column values are referenced using the dot notation

```
cursor_record_name.column_name |
cursor_record_name.alias_name
```

- Cursor attributes
  - `cursor_name%FOUND` and `cursor_name%NOTFOUND`
    - Indicates whether a row was fetched
  - `cursor_name%ROWCOUNT`
    - Maintains a count of the number of rows fetched from the cursor
  - `cursor_name%ISOPEN`
    - Indicates whether the cursor is open

Cursor Example

- Change the employee's area code from 212 to 818

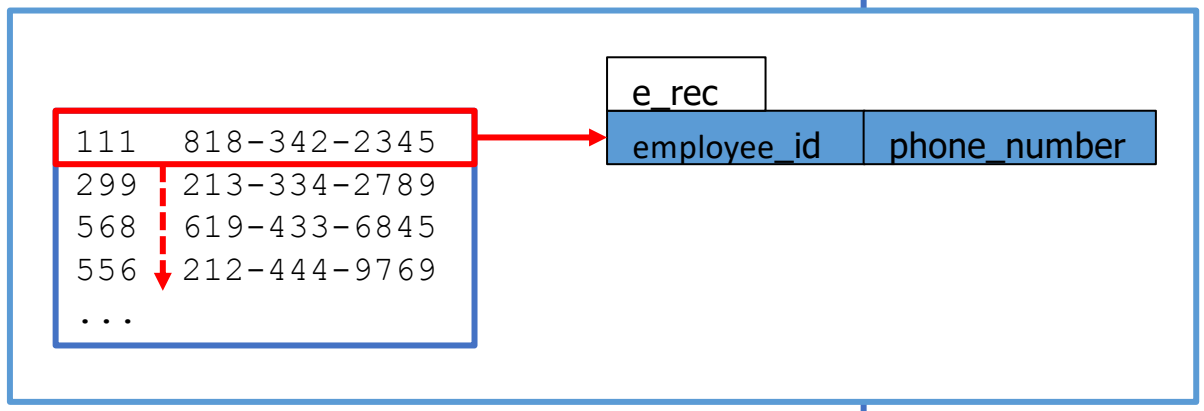
```
DECLARE
    CURSOR e_cur
    IS
        SELECT employee_id, phone_number
        FROM   employees;
    e_rec  e_cur%ROWTYPE;
BEGIN
    OPEN e_cur;

    LOOP
        FETCH e_cur INTO e_rec;
        EXIT WHEN e_cur%NOTFOUND;

        IF SUBSTR(e_rec.phone_number, 1,3) = '212' THEN
            e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

            UPDATE employees
            SET phone_number = e_rec.phone_number
            WHERE employee_id = e_rec.employee_id;
        END IF;
    END LOOP;

    CLOSE e_cur;
END;
```



Cursor Parameters

- A cursor can accept input parameters
  - Used to pass information to the cursor
    - Datatype cannot have a length qualifier

```
CURSOR cursor_name(parm datatype, ..., parm datatype)
IS
    select_statement
```

- Passed to the cursor as part of the `OPEN` command

```
OPEN cursor_name(parameter, ..., parameter);
```

Cursor Parameters Example

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT employee_id, phone_number
        FROM   employees
        WHERE  SUBSTR(phone_number, 1,3) = in_area_code;
    e_rec e_cur_parm%ROWTYPE;
BEGIN
    OPEN e_cur_parm('212');

    LOOP
        FETCH e_cur_parm INTO e_rec;
        EXIT WHEN e_cur_parm%NOTFOUND;
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE employee_id = e_rec.employee_id;
    END LOOP;
    CLOSE e_cur_parm;
END;
```

FOR-LOOP Cursors

- Simplifies the use of cursors
  - Automatically provides cursor management
    - Cursor record is implicitly declared
    - Opens the cursor when the loop is begun
    - Fetches one record for each loop iteration
    - Closes the cursor when the loop is complete
- Syntax:

```
FOR cursor_record IN cursor_name(parameter, ..., parameter)
LOOP
   -- Cursor processing statements;
END LOOP;
```

- Cursor record
  - Used as the loop counter
  - Available only inside the FOR-LOOP

FOR-LOOP Cursor Example

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT employee_id, phone_number
        FROM   employees
        WHERE  SUBSTR(phone_number, 1, 3) = in_area_code;
BEGIN
    FOR e_rec IN e_cur_parm('212') LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE employee_id = e_rec.employee_id;
    END LOOP;
END;
```

Using ROWID in Cursors

- Retrieve ROWID, then use it in the update instead of employee_id
  - Eliminates an index search by using ROWID to find the row

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT ROWID AS e_rowid, phone_number
        FROM   employees
        WHERE  SUBSTR(phone_number, 1, 3) = in_area_code;
BEGIN
    FOR e_rec IN e_cur_parm('212') LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE ROWID = e_rec.e_rowid;
    END LOOP;
END;
```

FOR UPDATE and CURRENT OF Clauses

- FOR UPDATE can be used to lock all cursor rows during the OPEN statement
  - Eliminates the possibility of waiting during the update, but may reduce concurrency

```
DECLARE
    CURSOR e_cur_parm(in_area_code VARCHAR2)
    IS
        SELECT phone_number
        FROM employees
        WHERE SUBSTR(phone_number, 1, 3) = in_area_code
        FOR UPDATE;
BEGIN
    FOR e_rec IN e_cur_parm('212') LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE CURRENT OF e_cur_parm;
    END LOOP;
END;
```

CURRENT OF is only permitted in FOR UPDATE cursors (no performance impact, but makes code easier to read)

Looping Over Implicit Cursors

- The `FOR LOOP` construct also works with implicit cursors
  - Does not allow `WHERE CURRENT OF`
  - Cannot refer to any cursor properties (such as `%NOTFOUND`)

```
BEGIN
    FOR e_rec IN (
        SELECT ROWID, phone_number
        FROM   employees
        WHERE  SUBSTR(phone_number, 1, 3) = '212'
    ) LOOP
        e_rec.phone_number := '818' || SUBSTR(e_rec.phone_number, 4);

        UPDATE employees
        SET phone_number = e_rec.phone_number
        WHERE ROWID = e_rec.ROWID;
    END LOOP;
END;
```

Exercise 9.2: Using Cursors

20 min

- Please complete this exercise in your Exercise Manual

BULK COLLECT

- Although the FOR LOOP cursor is easy to use, it is not the most efficient
  - BULK COLLECT in combination with FORALL is more efficient
- Requires a PL/SQL collection
  - For example, a variable size array (variable, but the max size is fixed)
    - `TYPE emp_array IS VARRAY(batchsize) OF cur%ROWTYPE;`
  - And a variable of that type
- BULK COLLECT comes in two forms:
  - SELECT … BULK COLLECT INTO collection
    - Useful if the total data volume is relatively small
  - FETCH … BULK COLLECT INTO collection [LIMIT batch_size]
    - Allows the data to be processed in batches
- FORALL allows a DML command to be executed for each item in a collection

BULK COLLECT Example

```
DECLARE
    CURSOR cur(in_area_code VARCHAR2) IS
        SELECT ROWID, phone_number FROM employees WHERE SUBSTR(phone_number, 1, 3) =
in_area_code;
    batchsize CONSTANT PLS_INTEGER := 1000;
    TYPE emp_array IS VARRAY(batchsize) OF cur%ROWTYPE;
    emps emp_array;
BEGIN
    OPEN cur(in_area_code => '818');
    LOOP
        FETCH cur BULK COLLECT INTO emps LIMIT batchsize;
        -- The for loop doesn't run when emps.COUNT() = 0
        FOR j IN 1..emps.COUNT() LOOP
            emps(j).phone_number := '515' || SUBSTR(emps(j).phone_number, 4);
        END LOOP;

        FORALL i IN 1..emps.COUNT()
            UPDATE employees e
            SET    e.phone_number = emps(i).phone_number
            WHERE  ROWID  = emps(i).ROWID;

        EXIT WHEN cur%NOTFOUND;
    END LOOP;
    CLOSE cur;
END;
```

Important not to test this before processing the batch, but also recognize that the last batch may be empty

Chapter Concepts

Working with PL/SQL

Control Structures and Exceptions

Cursors

Chapter Summary

Chapter Summary

In this chapter, we have discussed:

- Overview of PL/SQL
- Declaring, assigning, and using scalar variables
- Defining and using conditional, iterative, and sequential control
- Utilizing error handling and writing exception handlers
- Processing result sets with cursors
- Improving cursor processing with `FOR-LOOP` cursors
- Improving update and delete performance by using cursors