Lab 4. General Programming Language Fundamentals. SQL in PL/SQL

Databases II

4.1. PL/SQL Programming Fundamentals

CHARACTER TYPES

The PL/SQL engine accepts four types of characters: letters, digits, symbols (*, +, –, =, and so on), and white space. When elements from one or more of these character types are joined, they create a lexical unit (these lexical units can be a combination of character types). The lexical units are the words of the PL/SQL language.

LEXICAL UNITS

A language such as English contains different parts of speech. Each part of speech, such as a verb or noun, behaves in a different way and must be used according to specific rules. Likewise, a programming language has lexical units that are the building blocks of the language. PL/SQL lexical units fall within one of the following five groups:

- Identifiers must begin with a letter and may be up to 30 characters long.
- Reserved words are words that PL/SQL saves for its own use (such as BEGIN, END, and SELECT).
- Delimiters are characters that have special meaning to PL/SQL, such as arithmetic operators and quotation marks.
- Literals are values (character, numeric, or Boolean [true/false]) that are not identifiers. example: 123, “Declaration of Independence,” and FALSE
- Comments can be either single-line comments (--) or multiline comments (/* */).

THE MOST COMMON DATATYPES

The following are the major datatypes in Oracle that you can use in your PL/SQL:

1. VARCHAR2(*maximum_length*)
   - Stores variable-length character data.
   - Takes a required parameter that specifies a maximum length up to 32,767 bytes.

- Does not use a constant or variable to specify the maximum length; an integer literal must be used.
- The maximum width of a VARCHAR2 database column is 4,000 bytes.

2. CHAR[(*maximum_length*)]
    - Stores fixed-length (blank-padded if necessary) character data.
    - Takes an optional parameter that specifies a maximum length up to 32,767 bytes.
    - Does not use a constant or variable to specify the maximum length; an integer literal must be used. If the maximum length is not specified, it defaults to 1.
    - The maximum width of a CHAR database column is 2,000 bytes; the default is 1 byte.

3. NUMBER[(*precision*, *scale*)]
    - Stores fixed or floating-point numbers of virtually any size.
    - The precision is the total number of digits.
    - The scale determines where rounding occurs.
    - It is possible to specify precision and omit scale, in which case scale is 0 and only integers are allowed.
    - Constants or variables cannot be used to specify the precision and scale; integer literals must be used.
    - The maximum precision of a NUMBER value is 38 decimal digits.
    - The scale can range from 0 to 127. For instance, a scale of 2 rounds to the nearest hundredth (3.456 becomes 3.46).
    - The scale can be negative, which causes rounding to the left of the decimal point. For example, a scale of –3 rounds to the nearest thousandth (3,456 becomes 3,000). A scale of 0 rounds to the nearest whole number. If you do not specify the scale, it defaults to 0.

4. BINARY_INTEGER
    - Stores signed integer variables.
    - Compares to the NUMBER datatype. BINARY_INTEGER variables are stored in binary format, which takes up less space.
    - Calculations are faster.
    - Can store any integer value in the range –2,147,483,747 to 2,147,483,747.
    - This datatype is used primarily to index a PL/SQL table.

5. DATE
   - Stores fixed-length date values.
   - Valid dates for DATE variables are January 1, 4712 BC to December 31, 9999 AD.
   - When stored in a database column, date values include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight.
   - Dates are actually stored in binary format and are displayed according to the default format.

6. TIMESTAMP
   - This datatype is an extension of the DATE datatype. It stores fixed-length date values with a precision down to a fraction of a second, with up to nine places after the decimal (the default is six). An example of the default for this datatype is 12-JAN-2008 09.51.44.000000 PM.
   - The WITH TIME ZONE or WITH LOCAL TIME ZONE option allows the TIMESTAMP to be related to a particular time zone. Then this is adjusted to the time zone of the database. For example, this would allow a global database to have an entry in London and New York recorded as being the same time, even though it would be displayed as noon in New York and 5 p.m. in London.

7. BOOLEAN
   - Stores the values TRUE and FALSE and the nonvalue NULL. Recall that NULL stands for a missing, unknown, or inapplicable value.
   - Only the values TRUE and FALSE and the nonvalue NULL can be assigned to a BOOLEAN variable.
   - The values TRUE and FALSE cannot be inserted into a database column.

8. LONG
   - Stores variable-length character strings.
   - The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 2 gigabytes (GB).
   - You cannot select a value longer than 4,000 bytes from a LONG column into a LONG variable.
   - LONG columns can store text, arrays of characters, or even short documents. You can reference LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, SQL function calls, or certain SQL clauses, such as WHERE, GROUP BY, and CONNECT BY.

9. LONG RAW
   - Stores raw binary data of variable length up to 2GB.

10. LOB (large object)
    - The four types of LOBs are BLOB, CLOB, NCLOB, and BFILE. These can store binary objects, such as image or video files, up to 4GB in length.
    - A BFILE is a large binary file stored outside the database. The maximum size is 4GB.

11. ROWID
    - Internally, every Oracle database table has a ROWID pseudocolumn, which stores binary values called rowids.
    - Rowids uniquely identify rows and provide the fastest way to access particular rows.
    - Use the ROWID datatype to store rowids in a readable format.
    - When you select or fetch a rowid into a ROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value into an 18-byte character string and returns it in that format.
    - Extended rowids use base 64 encoding of the physical address for each row. The encoding characters are A to Z, a to z, 0 to 9, +, and /. ROWID is as follows: OOOOOOFFFBBBBBBRRR. Each component has a meaning. The first section, OOOOOO, signifies the database segment. The next section, FFF, indicates the tablespace-relative datafile number of the datafile that contains the row. The following section, BBBBBB, is the data block that contains the row. The last section, RRR, is the row in the block (keep in mind that this may change in future versions of Oracle).

OPERATORS (DELIMITERS): THE SEPARATORS IN AN EXPRESSION

The following are the operators you can use in PL/SQL:

- Arithmetic (** ,* ,/ ,+ ,–)
- Comparison (=, <>, !=, <, >, <=, >=, LIKE, IN, BETWEEN, IS NULL, IS NOT NULL, NOT IN)
- Logical (AND, OR, NOT)
- String (||, LIKE)
- Expressions
- Operator precedence

VARIABLE INITIALIZATION WITH SELECT INTO

PL/SQL has two main methods of giving value to variables in a PL/SQL block. The first one, is initialization with the `:=` syntax. In this lab you will learn how to initialize a variable with a select statement by using the SELECT INTO syntax.

A variable that has been declared in the declaration section of the PL/SQL block can later be given a value with a select statement. The syntax is as follows:

```
SELECT item_name INTO variable_name FROM table_name;
```

It is important to note that any single row function can be performed on the item to give the variable a calculated value.

Example:

```
SET SERVEROUTPUT ON
DECLARE
   v_average_cost VARCHAR2(10);
BEGIN
   SELECT TO_CHAR(AVG(cost), '$9,999.99') INTO v_average_cost FROM
   course;
   DBMS_OUTPUT.PUT_LINE('The average cost of a '||
      'course in the CTA program is '|| v_average_cost);
END;
```

In this example, a variable is given the value of the average cost of a course in the course table. First, the variable must be declared in the declaration section of the PL/SQL block. In this example, the variable is given the datatype of VARCHAR2(10) because of the functions used on the data. The same select statement that would produce this outcome in SQLDeveloper is as follows:

```
SELECT TO_CHAR(AVG(cost), '$9,999.99') FROM course;
```

The TO_CHAR function is used to format the cost; in doing this, the number datatype is converted to a character datatype. As soon as the variable has a value, it can be displayed to the screen in SQLDeveloper using the PUT_LINE procedure of the DBMS_OUTPUT package.

Data Definition Language (DDL) is not valid in a simple PL/SQL block. (More-advanced techniques such as procedures in the DBMS_SQL package enable you to make use of DDL.) However, DML is easily achieved either by use of variables or by simply putting a DML statement into a PL/SQL block. Here is an example of a PL/SQL block that UPDATEs an existing entry in the zip code table:

Example:

```
DECLARE
   v_city zipcode.city%TYPE;
BEGIN
   SELECT 'COLUMBUS' INTO v_city FROM dual;
   UPDATE zipcode SET city = v_city WHERE ZIP = 43224;
END;
```

It is also possible to insert data into a database table in a PL/SQL block, as shown in the following example:

Example:

```
DECLARE
   v_zip zipcode.zip%TYPE;
   v_user zipcode.created_by%TYPE;
   v_date zipcode.created_date%TYPE;
BEGIN
   SELECT 43438, USER, SYSDATE INTO v_zip, v_user, v_date
       FROM dual;
   INSERT INTO zipcode
      (ZIP, CREATED_BY, CREATED_DATE, MODIFIED_BY, MODIFIED_DATE)
      VALUES(v_zip, v_user, v_date, v_user, v_date);
END;
```

A PL/SQL block that inserts a new student in the STUDENT table:

```
DECLARE
   v_max_id number;
BEGIN
   SELECT MAX(student_id) INTO v_max_id FROM student;
   INSERT into student
       (student_id, last_name, zip,created_by, created_date,
        modified_by, modified_date,registration_date)
       VALUES (v_max_id + 1, 'Rosenzweig', 11238, 'BROSENZ ',
        '01-JAN-99','BROSENZ', '01-JAN-99', '01-JAN-99');
END;
```

To generate a unique ID, the maximum `student_id` is selected into a variable and then is incremented by 1. It is important to remember in this example that there is a foreign key on the zip item in the student table. This means that the zip code you choose to enter must be in the ZIPCODE table.

ORACLE SEQUENCE

An Oracle sequence is an Oracle database object that can be used to generate unique numbers. You can use sequences to automatically generate primary key values. After a sequence has been created, you can access its values in SQL statements with these pseudocolumns:

- CURRVAL returns the current value of the sequence.
- NEXTVAL increments the sequence and returns the new value.

The following statement creates the sequence ESEQ:

Example:

```
CREATE SEQUENCE eseq INCREMENT BY 10
```

The first reference to ESEQ.NEXTVAL returns 1.The second returns 11.Each subsequent reference returns a value 10 greater than the preceding one.

DRAWING NUMBERS FROM A SEQUENCE

You can insert a sequence value directly into a table without first selecting it.

The following example uses a table called test01. First the table test01 is created, and then the sequence `test_seq`, and then the sequence is used to populate the table.

Example:

```
CREATE TABLE test01 (col1 number);
CREATE SEQUENCE test_seq INCREMENT BY 5;
BEGIN
   INSERT INTO test01 VALUES(test_seq.NEXTVAL);
END;

/
Select * FROM test01;
```

A PL/SQL block that inserts a new student in the STUDENT table that uses the existing `student_id_seq` sequence to generate a unique ID for the new student.

Example:

```
DECLARE
```

```
   v_user student.created_by%TYPE;
   v_date student.created_date%TYPE;
BEGIN
   SELECT USER, sysdate INTO  v_user, v_date FROM dual;
   INSERT INTO student
     (student_id, last_name, zip,created_by, created_date,
      modified_by, modified_date, registration_date)
     VALUES (student_id_seq.nextval, 'Smith',11238, v_user,
             v_date, v_user, v_date, v_date);
END;
```

In the declaration section of the PL/SQL block, two variables are declared. They are both set to be datatypes within the student table using the `%TYPE` method of declaration. This ensures that the datatypes match the columns of the tables into which they will be inserted. The two variables `v_user` and `v_date` are given values from the system by means of SELECT INTO. The value of `student_id` is generated by using the next value of the `student_id_seq` sequence.

4.3 Making Use of SAVEPOINT

Transactions are a means to break programming code into manageable units. Grouping transactions into smaller elements is a standard practice that ensures that an application saves only correct data. Initially, any application must connect to the database to access the data. When a user issues DML statements in an application, the changes are not visible to other users until a COMMIT or ROLLBACK has been issued. Oracle guarantees a read-consistent view of the data. Until that point, all data that has been inserted or updated is held in memory and is available only to the current user. The rows that have been changed are locked by the current user and are not available for other users to update until the locks have been released. A COMMIT or ROLLBACK statement releases these locks. Transactions can be controlled more readily by marking points of the transaction with the SAVEPOINT command.

- COMMIT makes events within a transaction permanent.
- ROLLBACK erases events within a transaction.

Additionally, you can use a SAVEPOINT to control transactions. Transactions are defined in the PL/SQL block from one SAVEPOINT to another. The use of the SAVEPOINT command allows you to break your SQL statements into units so that in a given PL/SQL block, some units can be *committed* (saved to the database) and some can be *rolled back* (undone), and so forth.
To demonstrate the need for transaction control, we will examine a two-step data-manipulation process. For example, suppose that the fees for all courses in the Student database that had a prerequisite course needed to be increased by 10 percent. At the same time, all courses that did not have a prerequisite needed to be decreased by 10 percent. This is a two-step process. If one step was successful but the second step was not, the data concerning course cost would be inconsistent in the database. Because this adjustment is based on a change in percentage, there would be no way to track what part of this course adjustment was successful and what was not.
In the next example, you see one PL/SQL block that performs two updates on the cost item in the course table. In the first step (this code is commented to emphasize each update), the cost is updated with a cost that is 10 percent less whenever the course does not have a prerequisite. In the second step, the cost is increased by 10 percent when the course has a prerequisite.

Example:

```
BEGIN
-- STEP 1
   UPDATE course
      SET cost = cost  - (cost * 0.10)
    WHERE prerequisite IS NULL;
-- STEP 2
   UPDATE course
      SET cost = cost  + (cost * 0.10)
    WHERE prerequisite IS NOT NULL;
END;
```

Let's assume that the first update statement succeeds, but the second update statement fails because the network went down. The data in the course table is now inconsistent, because courses with no prerequisite have had their cost reduced, but courses with prerequisites have not been adjusted. To prevent this sort of situation, statements must be combined into a trans- action. So, either both statements will succeed, or both statements will fail.
A transaction usually combines SQL statements that represent a logical unit of work. The transaction begins with the first SQL statement issued after the previous transaction, or the first SQL statement issued after you connect to the database. The transaction ends with the COMMIT or ROLLBACK statement.

COMMIT

When a COMMIT statement is issued to the database, the transaction has ended, and the following results are true:

- All work done by the transaction becomes permanent.
- Other users can see changes in data made by the transaction.
- Any locks acquired by the transaction are released.

A COMMIT statement has the following syntax:

```
COMMIT [WORK];
```

The word WORK is optional and is used to improve readability. Until a transaction is committed, only the user executing that transaction can see changes in the data made by his or her session.

Suppose User A issues the following command on a STUDENT table that exists in another schema but that has a public synonym of student:

Example:

```
INSERT INTO student
```

```
        (student_id, last_name, zip, registration_date,created_by,
         created_date, modified_by,modified_date)
        VALUES (student_id_seq.nextval, 'Tashi', 10015,'01-JAN-99',
                'STUDENTA', '01-JAN-99','STUDENTA', '01-JAN-99');
```

Then User B enters the following command to query a table known by its public synonym STUDENT, while logged on to his session:

```
SELECT * FROM student WHERE last_name = 'Tashi';
```

Then User A issues the following command:

```
COMMIT;
```

If User B enters the same query again, he doesn't see the same results.

In this next example, there are two sessions: User A and User B. User A inserts a record into the STUDENT table. User B queries the STUDENT table but does not get the record that was inserted by User A. User B cannot see the information because User A has not committed the work. When User A commits the transaction, User B, upon resubmitting the query, sees the records inserted by User A.

ROLLBACK

When a ROLLBACK statement is issued to the database, the transaction has ended, and the following results are true:

. All work done by the transaction is undone, as if it hadn't been issued.
. Any locks acquired by the transaction are released.

A ROLLBACK statement has the following syntax:

```
ROLLBACK [WORK];
```

The WORK keyword is optional and is available for increased readability.

SAVEPOINT

The ROLLBACK statement undoes all the work done by the user in a specific transaction. With the SAVEPOINT command, however, only part of the transaction can be undone. The SAVEPOINT command has the following syntax:

```
SAVEPOINT name;
```

The word `name` is the SAVEPOINT's name. As soon as a SAVEPOINT is defined, the program can roll back to the SAVEPOINT. A ROLLBACK statement, then, has the following syntax:

```
ROLLBACK [WORK] to SAVEPOINT name;
```

When a ROLLBACK to SAVEPOINT statement is issued to the database, the following results are true:

. Any work done since the SAVEPOINT is undone. The SAVEPOINT remains active, however, until a full COMMIT or ROLLBACK is issued. It can be rolled back to again if desired.
. Any locks and resources acquired by the SQL statements since the SAVEPOINT are released.
. The transaction is not finished, because SQL statements are still pending.

A Single PL/SQL block can contain multiple transactions, as shown in this example:

Example:

```
DECLARE
   v_Counter NUMBER;
BEGIN
   v_counter := 0;
   FOR i IN 1..100
   LOOP
      v_counter := v_counter + 1;
      IF v_counter = 10
      THEN
        COMMIT;
        v_counter := 0;
      END IF;
   END LOOP;
END;
```

In this example, as soon as the value of `v_counter` becomes equal to 10, the work is committed. So, a total of 10 transactions are contained in this one PL/SQL block.

Exercises:

1. Write a PL/SQL block
   a. That includes declarations for the following variables:
      A VARCHAR2 datatype that can contain the string 'Introduction to Oracle PL/SQL'
      A NUMBER that can be assigned 987654.55, but not 987654.567 or 9876543.55
      A CONSTANT (you choose the correct data type) that is auto-initialized to the value '603D'
      A BOOLEAN
      A DATE data type autoinitialized to one week from today
   b. In the body of the PL/SQL block, put a DBMS_OUTPUT.PUT_LINE message for each of the variables that received an autoinitialization value.
   c. In a comment at the bottom of the PL/SQL block, state the value of your NUMBER data type.

2. Alter the PL/SQL block you created in Project 1 to conform to the following specifications:
    a. Remove the DBMS_OUTPUT.PUT_LINE messages.
    b. In the body of the PL/SQL block, write a selection test (IF) that does the following (use a nested IF statement where appropriate):
        i. Checks whether the VARCHAR2 you created contains the course named ‘Introduction to Underwater Basketweaving’.
        ii. If it does, put a DBMS_OUTPUT.PUT_LINE message on the screen that says so.
        iii. If it does not, test to see if the CONSTANT you created contains the room number 603D.
        iv. If it does, put a DBMS_OUTPUT.PUT_LINE message on the screen that states the course name and the room number that you’ve reached in this logic.
        v. If it does not, put a DBMS_OUTPUT.PUT_LINE message on the screen that states that the course and location could not be determined.

    c. Add a WHEN OTHERS EXCEPTION that puts a DBMS_OUTPUT.PUT_LINE message on the screen that says that an error occured.

3. Create a table called Ex3 with two columns; one is ID (a number) and the other is NAME, which is a VARCHAR2(20). Create a sequence called Ex3_SEQ that increments by units of 5.
4. Write a PL/SQL block that does the following, in this order:
    a. Declares two variables: one for `v_name` and one for `v_id`. The `v_name` variable can be used throughout the block to hold the name that will be inserted; realize that the value will change in the course of the block.
    b. The block inserts into the table the name of the student who is enrolled in the most classes and uses a sequence for the ID. Afterward there is SAVEPOINT A.
    c. The student with the fewest classes is inserted. Afterward there is SAVEPOINT B.
    d. The instructor who is teaching the most courses is inserted in the same way. Afterward there is SAVEPOINT C.
    e. Using a SELECT INTO statement, hold the value of the instructor in the variable `v_id`.
    f. Undo the instructor insertion by using rollback.
    g. Insert the instructor teaching the fewest courses, but do not use the sequence to generate the ID. Instead, use the value from the first instructor, whom you have since undone.
    h. Insert the instructor teaching the most courses, and use the sequence to populate his or her ID.

Add DBMS_OUTPUT throughout the block to display the values of the variables as they change.