PL/SQL: Introduction

Overview

PL/SQL is a programming language-like extension of SQL. Its implementation is available on Oracle DBMS servers.

Features of PL/SQL:

- Procedural extension of SQL.
- Variables
- Assignments
- Program control (if-then-else, loops)
- Use of SQL update and query statements
- Cursors, rows, tables
- Anonymous blocks, procedures, functions
- Compiled
  - needs static database schema
  - no database schema alteration (DDL) commands
  - stored procedures
- library packages
  - I/O
  - dynamic schemas (allows for use of DDL commands)

Anonymous Blocks and Basic Language Features

An anonymous block is the most basic type of PL/SQL program. The syntax of an anonymous block is

```
DECLARE
   <declaration-section>

BEGIN
   <executable-section>

[EXCEPTION
   <exception-handling section>]
END;
```

Declaration Section, Variables, Type System

The content of the declaration section consists of different declarations. All declarations must end with a semicolumn (";"). The following declaration types exist in SQL/PL.

Constant declaration. All constants used in the anonymous block must be declared. The syntax of a constant declaration is

```
<ConstantName> constant <Type> := <Value>;
```

`<ConstantName>`:	name of the constant;
`<Type>`:	type of the constant (see below);
`<Value>`:	value of the constant;

Variable declaration. All global variables in PL/SQL must be declared in the declaration section[1]. The format of the variable declaration is

```
<VarName> <Type> [not null] [:= <Value>];
```

`<VarName>`:	name of the variable;
`<Type>`:	type of the variable (see below);
`not null`:	(optional) if present, variable must be initialized and may not be `null`
`<Value>`:	(optional) initial value of the variable.

Procedure declaration. SQL/PL procedures are similar to procedures in Pascal, or C/Java `void` functions/methods. Procedures are discussed below.

Function declaration. SQL/PL functions have a return value. Functions are discussed below.

Type declaration. Declarations of user-defined types in SQL/PL. Discussed in detail below.

Atomic Data Types

SQL/PL has a type system that is **different from the type system of SQL**. SQL types have corresponding SQL/PL type, but the names of the types may differ.

[1] SQL/PL allows ad-hoc use of variables with local scope in loops. Such variables need not be declared at the beginning of the block.

SQL/PL uses the following atomic types.

Category	SQL/PL Type	SQL Type	Explanation
Numeric	`binary_integer`	`INT`	
	`natural`		unsigned integer, 0 to $2^{3}1$
	`positive`		positive integer, 1 to $2^{3}1$
	`number(m,n)`	`NUMBER(m,n)`	
Character	`char(n)`	`CHAR(n)`	
	`varchar2(n)`	`VARCHAR2(n)`	
Boolean	`boolean`	`BOOLEAN`	
Date/Time	`date`	`DATE`	

Note: a single variable or constant declaration declares exactly one variable or constant.

Anchored Types

SQL/PL allows to declare that a type of a variable or constant shall be the same as the type of another variable, or, more importantly, the type of a column from some relational table in the database. The *type anchoring* operator % is used to specify the anchored type as follows:

```
<ConstantName>  constant <Variable-or-TableColumn>%type := <Value>;

<VarName>  <Variable-or-TableColumn>%type [not null] [:= <Value>];
```

Examples

Below is an example of a declaration section of an anonymous block:

```
DECLARE
                      -- declaration section starts here
 MaxCapacity constant natural := 35;                   -- nonnegative integer constant
 classSize natural;                                    -- nonnegative variable
 studentId natural :=0;                                -- nonnegative initialized variable
 databaseCourse constant Courses.Name%type = 'CPE 365';   -- constant with type anchored to Courses.Name
 currentCourse  Courses.Name%type;                     -- variable with type anchored to Courses.Name
 anotherCourse  currentCourse%type;                    -- variable with type anchored to another variable
```

Comments. PL/SQL uses `--` to indicate that the rest of the line is a comment. Multiline comments are enclosed in `/* ...*/`.

Executable Section, Statements

The *executable section* of an anonymous block (as well as the bodies of procedures and functions) consists of a sequence of PL/SQL statements.

The following statements are defined in PL/SQL:

Statement type	Syntax	Explanation
Null statement	null;	no operation
Assignment	<Var>:=<expression>;	assignment
Conditional	if <condition> then <statements>; end if;	single condition
Conditional	if <condition> then <statements>; else <statements> end if;	if-then-else
Conditional	if <condition> then <statements>; elsif <condition> then <statements>; ... else <statements>; end if;	conditional with multiple cases
Conditional	case <expression> when <value> then <statements>; ... [else <statements>;] end case	case statement
Conditional	case when <booleanExp> then <statements>; ... [else <statements>;] end case	searched case statement
Exit	exit [when <expression>];	exit from current block statement (loop)
Loop	loop <statements>; end loop	simple loop w/o termination
Loop	for <loopCounter> in [reverse] <lower>..<upper> loop <statements>; end loop;	For-loop
Loop	while <condition> loop <statements>; end loop;	While-loop
Loop	for <recordInedx> in <Cursor> loop <statements>; end loop;	Foreach loop for database table cursors
Procedure call	<ProcedureName>(<Parameters>);	call specified procedure with specified parameters
SQL Statement	<SQL Statement>;	execute the SQL statement

Database Access

When PL/SQL program is running it:

- has access to all available database tables;
- may include SQL statements that query, modify the tables (but not the database schema);

SQL/PL has three compound data types to facilitate database access:

- cursors
- records
- tables.

(Explicit) Cursors

A cursor is a class of data types designed to store a collection (sequence) of table rows. All global cursors have to be specified in the declaration portion of an anonymous block.

Cursors are declared as follows:

```
cursor <CursorName> [return <type>] is <SelectStatement>;
```

<CursorName>:	name of the cursor
<type>:	type of the single row in the cursor (optional)
<SelectStatement>:	SQL SELECT statement defining the cursor contents

Note:

- cursors get their values defined at the beginning of the program, **before** and statements in the body of the block are executed;
- cursors cannot change their values from the result of one SELECT query to the result of another.

To define the type of the returned tuple, either row types or row type anchors are used. A row type anchor has syntax

```
<Table>%rowtype
```

where <Table> is the relational table whose tuple type serves as the anchor.

Example. Here are some cursor declarations.

```
cursor allCourses is SELECT Course.Name, Course.Num FROM Courses;

cursor allCSStudents return Students%rowtype is
        SELECT *
        FROM Students
        WHERE Major = 'CS';
```

Working with cursors

There are two ways to work with the contents of a cursor: manual navigation, and the cursor for-loop. Also, cursor variables have attributes that are accessible from within PL/SQL statements for assessment of the current state of the cursor.

Manual Navigation

Three statements support manual cursor navigation:

open <CursorName>; this statement results in the cursor being opened and the cursor pointer positioned at the first record of the cursor.

fetch <CurSorName> **into ¡Names¿;** fetch operation retrieves current tuple from the cursor and puts its contents into the variables specified in the <Names> list. alternatively, a single record-type variable name can be specified, in which case the tuple will be imported into it. The cursor pointer is moved to the next tuple.

close <CursorName>; this statement closes current cursor.

Cursosr attributes allow PL/SQL statements assess current state of the cursor. The attributes are accessed via the

```
<CursorName>%<CursorAttribute>
```

syntax. The attributes are:

- %found : true if the last fetch statement (see below) yielded a record.
- %notfound: true if the last fetch statement did not yield a record.
- %rowcount: total number of rows fetched from the cursor already.
- %isopen: true if the cursor is open.

Cursor for-loop

The syntax of the loop is:

```
for <recordIndex> in <CursorName>
  loop
    <statements>;
  end loop;
```

<recordIndex>:	a record-type local (loop) variable.
<CursorName> :	name of the cursor being iterated.

Parameterized Cursors

Cursors can parameterized. The syntax is

```
cursor <CursorName> (<parameters>)
 [return <type> ]
is <SelectStatement>;
```

<CursorName>:	name of the cursor
<type>:	type of the single row in the cursor (optional)
<SelectStatement>:	SQL SELECT statement defining the cursor contents
<parameters>:	list of the cursor's parameters

Each parameter is declared using the following syntax:

```
<ParameterName> [IN] <type> [{:= | DEFAULT} <expression>]
```

Here,

<ParameterName>:	name of the parameter
IN:	optional specification of the input parameter
<type>:	type of the parameter
DEFAULT:	specification of the default value
<expression>:	default or assigned value of the parameter

The cursor is parameterized in the open command:

```
open <cursorName>(<values>);
```

Example. An example of a parameterized cursor declaration is shown below:

```
declare

cursor mycursor(prefix char(4), cn  IN Courses.CourseNumber%type) is
           select s.* from Students s, Enrollments e
           where s.SSN = e.Student and
                 e.Course = (select courseId from Courses
                             where Dept = prefix and CourseNumber = cn);
```

This cursor results in a list of students enrolled in a course, specified by the course prefix-course number parameter pair. The cursor can then be instantiated in the body of the block:

```
open mycursor('CPE', 365);  // mycursor now contains the list of
                            // students in CPE 365
```

Cursor Variables

Cursor variables are different from explicit cursors. In particular:

- *Cursor variables* do not need to be associated with specific SELECT statements in the declaration portion of the block;
- Prior to declaring *cursor variables*, an appropriate *cursor reference type* must be declared.
- *Cursor variables* are actually *textbfpointers* to the locations of cursors.

The type declaration is

```
type <Name> is ref cursor [return <Type>];
```

Cursor variables are then declared as variables of type `<Name>`.

In the body of the block, cursor variables are instantiated using the following syntax of the `open` statement:

```
open <CursorVarName> for <SelectStatement>;
```

`<CursorVarName>`:	name of the cursor variable
`<SelectStaement>`:	SELECT statement to be used to populate the cursor

Example. We can use the cursor variable to build the cursor described in the previous example as follows:

```
declare

  type student_record_cursor is ref cursor;

  student_record_cursor  mycursor;

begin
   open mycursor for  select s.* from Students s, Enrollments e
```

```
                              where s.SSN = e.Student and
                              e.Course = (select courseId from Courses
                                          where Dept = 'CPE' and CourseNumber = 365);
...
end;
```

Records

SQL/PL record type family is designed to provide convenient way of storing individual database tuples. A record variable can appear in SQL/PL code via the following means:

- a user-defined record type was defined and a variable of this type declared;
- a variable was declared to have a type described by the `%rowtype` attribute of a database table, cursor or a table type.

The syntax of a user-defined type declaration is:

```
type <TypeName> is record (<fields>);
```

`<TypeName>`:	name of the record type
`<fields>`:	list of fields forming the record

The syntax for each field declaration is:

```
<Name> <type> [[not null] {:=|default} <expression>]
```

`<Name>`:	name of the field
`<type>`:	type of the field
`not null`:	indicates that the record cannot be null
`default` :	indicates that the record has a default value
`<expression>`:	assigned/default value of the record type

Field declarations are separated by commas.

Access to individual fields of the record is via the

```
<VarName>.<FieldName>
```

syntax.

Example The fragment below shows how to define and instantiate record type variables.

```
declare
  type student_record_type is record   // a container type
       SSN binary_int DEFAULT 0,       // for a student record
       firstName char(20),
       lastName char(20),
       major char(10) DEFAULT 'undeclared',
       year  binary_int;

  student1 student_record_type;       // two student record
  student2 student_record_type;       // variables
```

```
  course Courses%rowtype;             // a record variable for the Courses table

begin
   student1.SSN : = 123451234;
   student1.firstName := 'John';
   student1.lastName  := 'McCormick';
   student1.major := 'CSC';
   student1.year := 2005;

   student2 := student1;
   student2.firstName := 'Alice';
   student2.SSN := 223233322;

   course.Dept := 'CPE';
   course.CourseNumber := 365;

   ...
end;
```

Tables

In SQL/PL, *table type* defines *unbounded sparse single-column arrays*. (note, that the single column can be of record type). There are two types of tables: *index-by tables* and *nested tables*. We will mostly discuss the former.

The syntax of a *index-by table* type declaration is:

```
type <Name> is table of <type>
  index by binary_integer
```

`<Name>`: name of the table type
`<type>`: type of the entry of the table (can be atomic or record type)

Index-by tables index each table entry (row) using an integer index. When the table is created, indexes are dense, but they may become sparse with use.

The following operations can be performed on table variables:

- `<TableVar>(<Value>)`. Returns the element indexed at row `<Value>`.
- `<TableVar>.count`. Returns the number of elements (rows) in the table.
- `<TableVar>.delete(<Value>)`. Removes from the table the record indexed at location `<Value>`. Also possible `<TableVar>.delete(<Value1>,<Value2>)`: deletes all rows between (and including) `<Value1>` and `<Value2>`.
- `<TableVar>.exists(<Value>)`. Returns `true` if the table contains the row at the index `<Value>`.
- `<TableVar>.last`. Returns the highest-valued index from the table.
- `<TableVar>.next(<Value>)`. Returns the index following the `<Value>` parameter, which contains a table row.
- `<TableVar>.prior(<Value>)`. Returns the index preceding the `<Value>` parameter, which contains a table row.