SCSX1056 – ORACLE & SQL
UNIT V - CURSOR MANAGEMENT AND DATABASE TRIGGERS

Static Cursors – REF Cursors Subprograms: Procedures – Functions – Packages Database Triggers – Creating Triggers – Types – Built-in-packages

Cursor management

Oracle allocates an area of memory known as context area for the processing of SQL statements. The context area contains information necessary to complete the processing, including the number of rows processed by the statement, a pointer to the parsed representation of the statement.

A cursor is a handle or pointer to the context area. Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed. The three types of cursors are

- Static cursors
- Dynamic cursors
- REF cursors

Static cursors definitions are those whose select statements are known at compile time. These are further classified into:

- Explicit cursor
- Implicit cursor

An explicit cursor is one in which the cursor name is explicitly assigned to the select statement. An implicit cursor is used for all other SQL statements. Processing an explicit cursor involves four steps. Processing of an implicit cursor is taken care of by PL/SQL. The declaration of the cursor is done in the declarative part of the block.

Dynamic cursor is made possible in PL/SQL only through the use of DBMS_SQL built-in package.

A cursor variable is a reference type. A reference type is similar to a pointer. It can name different storage locations as the program runs. In order to use the reference type, the variable has to be declared and the storage has to be allocated. REF cursors are further classified based on the return type.

- Strong cursor
- Weak cursor

A strong cursor is a cursor whose return type is specified. A weak cursor is a cursor whose return type is not specified.

Static cursors

These are further classified into:

- Explicit cursor
- Implicit cursor

Explicit cursor

The set of rows returned by a query can contain zero or multiple rows depending upon the query defined. These rows are called the active set. The cursor will point to the current row in the active set.
After declaring a cursor, we can use the following commands to control the cursor.

- Open
- Fetch
- close

the 'open' statement executes the query, identifies the active set and positions the cursor before the first row. The syntax is given below

open <cursor_name>;

the fetch statement retrieves the current row and advances the cursor to the next row to fetch the remaining rows. Syntax for fetch is as given below:

fetch <cursor_name> into <column_name>;

after processing the last row in the active set, the cursor is disabled with the help of the 'close' command. The syntax is as follows:

close <cursor_name>;

example

```
declare
icode order_detail.itemcode%type;
cursor a is select itemcode from order_detail where itemcode='i201';
begin
  open a;
  loop
       fetch a into icode;
       update itemfile set itemrate=22.05 where icode=itemcode;
       exit when a%NOTFOUND;
       end loop;
      dbms_output.put_line('table updated');
close a;
end;
```

the output of the above coding is given below,

table updated.

PL/SQL procedure successfully completed.

Explicit cursor attributes when appended to the cursor name allow us to access useful information from the retrieved rows. They are

%notfound
%found
%rowcount
%isopen

%notfound

After opening a cursor, a 'fetch' statement is used to fetch rows from the active set, one at a time. The attribute %notfound indicates whether fetch statement returns row from the active set. If the last fetch fails to return a row, then %notfound evaluates to true, else, it evaluates to false.

Example

```
Declare
 Order_no order_detail.orderno%type;
 Cursor a is select orderno from order_detail where orderno='c001';
Begin
  Open a;
  Loop
    Fetch a into order_no;
    Update order_master set del_date=sysdate where orderno='c001';
  Exit when a%notfound;
 End loop;
End;
```

%found

The %found attribute is the logical opposite of %notfound. It evaluates to true if the last 'fetch' statement succeeds in returning a row. It would be evaluated to false if the last 'fetch' command failed because no more rows were available.

%rowcount

The %rowcount attribute is used to return the number of rows fetched. Before the first fetch, %rowcount is zero. When the 'fetch' statement returns a row, then the number is incremented.

Example

```
Declare
 Cursor a is select * from order_detail where orderno='c001';
  Myorder order_detail%rowtype;
Begin
  Open a;
```

```
 Loop
   Fetch a into myorder;
   exit when a%notfound;
   dbms_output.put_line('fetched' || a%rowcount || 'from table');
End loop;
End;
```

%isopen

if the cursor is already open, then, the attribute %isopen evaluates to true, else it evaluates to false.

Example

```
Declare
Cursor mycur is select * from order_master;
Begin
If not mycur%isopen then
   dbms_output.put_line('the cursor is yet to be opented');
end if;
open mycur;
if mycur%isopen then
   dbms_output.put_line('the cursor is now open');
end if;
close mycur;
end;
```

Implicit cursor

PL/SQL implicitly declares cursors for all SQL data manipulation statements, including queries that return one row. For queries that return more than one row, we should use explicit cursors to access the rows individually.

Implicit cursor attributes can be used to access information about the most recently executed SQL statement. The most recently executed SQL statement is referred as 'SQLCURSOR'. The implicit cursor attributes are:

- %notfound
- %found
- %rowcount
- %isopen

%notfound

The %notfound attribute evaluates to true if DML statements do not return any row, else it evaluates to false.

```
Begin
Delete from order_detail where orderno='o001';
```

```
If sql%notfound then
   dbms_output.put_line('value not found');
else
   dbms_output.put_line('value found and deleted');
end if;
end;
```

%found

the %found attribute is the logical opposite of the %notfound attribute. The %found attribute is evaluated to true if the SQL DML statement affects one or more rows, else it is evaluated to false.

%rowcount

the %rowcount attributes counts the number of rows returned by an SQL DML statement. The %rowcount will return zero if the DML statement does not affect any row.

```
Declare
Order_no order_master.orderno%type;
Begin
Select orderno into order_no from order_master where orderno='o0001';
If sql%rowcount > 0 then
dbms_output.put_line('rows selected from table');
else
dbms_output.put_line('no rows selected from table');
end if;
end;
```

%isopen

Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %isopen is always evaluated to false.

REF cursors

An explicit cursor is a static cursor i.e. the cursor is associated with one SQL statement and the statement is known when the block is compiled. A cursor variable can be associated with different statement at run time. Cursor variables are similar to PL/SQL variables, which can hold different values at run time. Static cursors are similar to PL/SQL constants, as they can be associated with only one run time query.

REF type

Where type is a previously defined type. The REF keyword indicates that the new type will be a pointer to the defined type. The type of the cursor is therefore a REF cursor. The complete syntax for defining a cursor variable type is,

Type type_name is ref cursor return return_type;

Where type_name is the name of the new reference type, and the return type is the record type indicating the types of the select list that will be returned by the cursor variable. The return type for a cursor variable must be record type. It can be declared explicitly as a user defined record, or implicitly using %rowtype. Once the reference type is defined, the variable can be declared.

Constrained and Unconstrained cursor variables

When the cursor variable, has a return type it is known as a constrained cursor variable or a strong cursor. However, cursor variables need not necessarily have a return type and such cursor variables are know as unconstrained cursor variables or weak cursors.

Type t_add_ref is ref cursor return vendo_master%rowtype;

Type t_adds_ref is ref cursor;

The first line gives us the declaration of a strong cursor and the second line gives the declaration of a weak cursor. Note that the return type is specified in the first declaration and is hence a strong cursor and a variable that is declared based on a strong cursor is known as a constrained cursor variable. In the second declaration the return type is not specified and is hence a weak cursor declaration and a cursor variable declared on this type of cursor is known as an unconstrained cursor variable. Cursor variables can be declared as shown

V_add_ref t_add_ref;

Opening a cursor variable for a query

A cursor variable must be associated with a particular select statement. This is achieved by the open syntax, which is extended to allow the query to be specified. Using the 'open for' syntax as shown below does this:

Open cursor_variable for select_statement;

Closing cursor variables

Cursor variables are closed with the close statement just like static cursors.

```
Declare
Type r1_cur is ref cursor;
Var1 r1_cur;
Nam varchar2(5);
No number(2);
```

```
Begin
No:=&enter_no;
If no=10 then
Open var1 for
Select orderno from order_master where vencode='voo1';
Fetch var1 into nam;
dbms_output.put_line('order no is ' || nam);
close var1;
else
open var1 for
select qty_ord from order_detail where orderno='o001';
loop
fetch var1 into  no;
exit when var1%notfound;
dbms_output.put_line('quantity ordered is ' || no);
end loop;
close var1;
end if;
end;
```

Subprograms

Subprograms are named PL/SQL blocks that can accept parameter can be invoked whenever required. Similar to a PL/SQL, a subprogram can also have a declarative part, an executable part and an exception handling part. Some of the important features offered by subprograms are given below:

- Modularity – subprograms allow us to break a program into manageable, well-defined logical modules.
- Reusability – subprograms once executed can be used in any number of applications.
- Maintainability – subprograms can simplify maintenance, because if a subprogram is affected, only its definition changes.

PL/SQL supports two types of subprograms. They are

- Procedures
- Functions

Procedures are usually used to perform any specific task and functions are used to compute a value.

Procedures

A procedure is a subprogram that performs a specific action. The syntax for creating a procedure is given below;

```
Create or replace procedure <proc_name> [parameter list] is <local declarations>;
Begin
(executable statement);
```

[exception] (exception handler)
end;

A procedure has two parts, namely, specification and body. The procedure specification begins with the keyword procedure and ends with the procedure name or parameter list. The procedure body begins with the keyword is and ends with the keyword end. It can also include declarative, executable and exceptional parts within the keywords are and end. Syntax to execute a procedure is given below.

Exec <proce_name> (parameters);

While declaring variables in the declarative part of the procedure body, we should not specify the width of the datatype.

Procedure width (name char(40)) is
Begin
(set of statements);
end;

in the above example, char(40) should be replaced by char. The example shown below explains the usage of a procedure. It accepts a single parameter and updates the table based on a condition. It also raises an exception if n o data is retrieved.

Create or replace procedure items (orders varchar2) is
Qtyhand number;
Relevel number;
Maxlevel number;
Begin
Select qty_hand,re_level,max_level into qtyhand,relevel,maxlevel from itemfile where itemcode=orders;
If qtyhand<relevel then
Update itemfile set qty_hand =relevel + qtyhand where itemcode=orders;
Else
dbms_output.put_line('itemlevel ok');
end if;
exception
when no_data_found then
dbms_output.put_line('no data returned');
end;

the above procedure items that has been created can be executed from the SQL prompt as shown below;

exec items('i201');

the parameters list (defined in the create procedure command) can hold any of the following mdoes, namely, in (by default), out and inout. These parameter modes can be used within any subprograms.

In parameter

The in parameter mode is used to pass values to the subprogram when invoked.

```
Create or replace procedure orders( a in varchar2) is
V_code varachar2(5);
O_stat char(1);
Begin
Select vencode,ostatus into v_code, o_stat form order_master where orderno = a;
If o_stat = 'p' then
dbms_output.put_line('pending order' || a);
else
dbms_output.put_line('completed order' || a);
end if;
end;
```

on compilation a message as shown below is displayed.

Procedure created.

The above procedure can be executed as shown below:

```
Exec orders('o001');
```

The output appears as shown below:

```
Completed order o001
PL/SQL procedure successfully completed.
```

Out parameter

The out parameter mode is used to return values to the caller of a subprogram. Since the initial value for an out parameter is undefined, its value can b e assigned to another variable.

```
Create or replace procedure test( a in varchar2, b out number) is identity number;
Begin
Select qty_ord into identity from order_detail where orderno = a;
If identity < 450 then
B:=100;
End if;
End;
```

The above procedure can be executed from another program, which will display the output of the variable b. the program to display the out parameter specified in the procedure is given below:

```
Declare
A varchar2(5);
B number;
Begin
```

```
Test('o202', b);
dbms_output.put_line('the value of b is ' || to_char(b));
end;
```

the output of the above program is
the value of b is 100

in out parameter

the in out parameter is used to pass initial values to the subprogram when invoked and it also returns updated values to the caller. An in out parameter acts like an initialized variable and, therefore, can be assigned to other variables or to itself.

```
Create or replace procedure or_detail (orno in varchar2, b in out varchar2) is
Qtyord number;
Qtydeld number;
Code varchar2(5);
Begin
Select qty_ord,qty_del,itemcode into qtyord,qtydeld,code from order_detail where orderno=orno;
If qtydeld < qtyord then
B:=code;
End if;
End;
```

The output of the above program is
Procedure created.

To execute the above procedure a block as shown below is written and executed.

```
Declare
A varchar2(5);
B varchar2(5);
Begin
Or_detail('o201',b);
dbms_output.put_line('the item code is ' || b);
end;
```

the output of the above program is
the item code is i201

Functions

A function is a subprogram that computes a value. The syntax for createing a function is given below:

```
Create or replace function <function_name> [argument]
Return datatype is
(local declaraction)
begin
(executable statements)
```

[exception]
end;

similar to procedure, a function also has tow parts, namely, the function specification and the founction body. The function specification begins with the keyword function and end with the return calue. The function body begins with the keyword is and ends with the keyword end.

```
create or replace function items(it varchar2)
return number is
args number;
qtyhand number;
relevel number;
maxlevel number;
begin
select qty_hand, re_level, max_level into qtyhand, relevel, maxlevel from itemfile
where itemcode = it;
if (qtyhand + relevel) > maxlevel then
args:=maxlevel;
relevel args;
end if;
end;
```

the output of the above block of code is
function created.

To execute the function items the following block of code is executed.

```
Declare
A varchar2(5);
B number;
Begin
A:=&a;
B:=item(a);
dbms_output.put_line('the value returned is ' || b);
end;
```

the output of the above block of code is
enter value for b: 'i202'
the value returned is 140

Packages

A package is a database object, which is an encapsulation of related PL/SQL types, subprograms, cursors, exceptions, variables and constants. It consists of two parts, a specifications and a body. In the package specification we can declare types, variables, constants, exceptions, cursors and subprograms. A package body implements cursors, subprograms defined in the package specification.

Packages can be created using the following commands

- Create package command
- Create package body command

The package specification is declared using a 'create package' command. The syntax for the 'create package' command is as follows.

Create package <package_name> is <declarations>
Begin
(executable statements)
end [package name];

the procedures and cursors declared in the 'create package' command is fully defined and implemented by the package body, which can b e achieved by using the following syntax

create package body <package_name> is <declarations>
begin
(executable statements)
end [body_name];

in the 'create package body' commands, the keywords, 'public' and 'private' denote the usage of object declaration in a package.

The package specification

The package specification contains public objects and types. It can also include subprograms. The specifications contain the package resources required for our applications.

Package body

The package body contains the definition of every cursor and subprogram declared in the package specification and implements them. Private declarations can also be included in a package body. The initialization part of the package body is optional, it may consist of statements that initialize some of the variables previously declared in the package. The initialization part of a package plays a minor role, because, neither can a package can be called nor parameters be passed to the package. Therefore, the initialization part of a package is run only once.

Create or replace package pack_me is
Procedure order_proc (orno varchar2);
Function order_fun(ornos varchar2) return varchar2;
End pack_me;

The package body is coded as given below:

Create or replace package pack_me is
Procedure order_proc (orno varchar2) is
Stat char(1);
Begin

```
Select ostatus into stat from order_master where orderno=orno;
If stat = 'p' then
dbms_output.put_line('pending order');
else
dbms_output.put_line('completed order');
end if;
end order_proc;
Function order_fun(ornos varchar2) return varchar2 is
Icode varchar2(5);
Ocode varchar2(5);
Qtyord number;
Qtydeld number;
Begin
Select qty_ord,qty_deld,itemcode,orderno into qtyord,qtydeld,icode,ocode from order_detail where orderno=orno;
If qtyord<qtydeld then
Return ocode;
Else
Return icode;
End if;
End order_fun;
End pack_me;
```

Calling packaged subprograms

To reference the types, objects and subprograms declared in a package specification the following notation is used.

Package-name.type-name
Package-name.object-name

To execute the function that is given in the package a block of code is written as shown below;

```
Declare
A varchar2(5);
B varchar2(5);
Begin
B:=pack_me.order_fun('o202');
dbms_output.put_line('the value is' || b);
end;
```

Database triggers

A database trigger is a stored procedure that is fired when an insert, update or delete statement is issued against the associated table. Database trigger can be used for the following purposes.

- To generate data automatically.

- To enforce complex integrity constraints. (e.g. checking with sysdate, checking with data in another table).
- To customize complex security authorizations.
- To maintain replicate tables
- To audit data modifications

Syntax for creating triggers

The syntax for creating a trigger is given below.

Create or replace trigger <trigger name> [before/after] [insert/update/delete] on <table name> [for each statement/for each row] [when <conditions>];

A database trigger can also have declarative and exception handling parts.

Parts of trigger

A database trigger has three parts, namely, a trigger statement, a trigger body and trigger restrictions.

Trigger statement

The trigger statement specifies the DML statements like update, delete and insert and it fires the trigger body. It also specifies the table to which the trigger is associated.

Trigger body

Trigger body is a PL/SQL block that is executed when a triggering statement is issued.

Trigger restriction

Restrictions on a trigger can be achieved using the WHEN clause as shown in the syntax for creating triggers. They can be included in the definition of a row trigger, wherein, the condition in the WHEN clause is evaluated for each row that is affected by the trigger.

Types of triggers

Triggers are categorized into the following types based on when they are fired:

- Before
- After
- For each row
- For each statement (default)

Before/after options

The before/after options can be used to specify when the trigger body should be fired with respect to the triggering statement. If the user includes a before option, then, oracle fires the trigger before executing the triggering statement. On the other hand, if AFTER is used, then, oracle fires the trigger after executing the triggering statement.

For each Row/statement

When the for each row/statement option when included in the 'create trigger' syntax specifies that the trigger fires once per row. By default, a database trigger fires for each statement.

```
Create or replace trigger orders
Before insert on order_detail for each row
Declare
Orno order_detail.orderno%type;
Begin
Select orderno into orno from order_detail where qty_ord<qty_deld;
If orno = 'o001' then
Raise_application_error (-20001,'enter some other number');
End if;
End;
```

A procedure named raise_application_error to issue user-defined error message. The syntax is given below
raise_application_error(error_number, ' error message');

the error_number ranges from -20000…. -20999 and error-message can be a character string.

Built-in packages

The database user SYS owns all the supplied packages. They are public synonyms and can be accessed by any user. EXECUTE permission on the package is necessary for users other than SYS to call the procedures and functions within the packages.

A list of the packages is shown below:

Package name	Description
DBMS_ALERT	Synchronous inter session communication
DBMS_APPLICATION_INFO	Allows registering of an application of tracing pruposes
DBMS_AQ & DBMS_AQADM	Management of oracle8 advanced queuing option
DBMS_DEFER, DBMS_DEFER_SYS & DBMS_DEFER_QUERY	Allows building and administering deferred remote procedure calls

DBMS_DDL	PL/SQL equivalents for some DDL commands
DBMS_DESCRIBE	Describes stored subprograms
DBMS_LOB	Manipulation of oracle8LOB
DBMS_JOB	Allows scheduling of PL/SQL procedures
DBMS_LOCK	User defined locks
DBMS_OUTPUT	Provides screen output in SQL *plus or server manager
DBMS_PIPE	Asynchronous inter session communication
DBMS_REFRESH& DBMS_SNAPSHOT	Allows managing or snapshots
DBMS_REPCAT, DBMS_REPCAT_AUTH& DBMS_REPCAT_ADMIN	Allows management of oracle's symmetric replication facility
DBMS_ROWID	Allows obtaining of information from a ROWID, and conversion between oracle7 and oracle 8 ROWIDs
DBMS_SESSION	PL/SQL equivalents for alter session
DBMS_SHARED_POOL	Control of the shared pool
DBMS_SQL	Dynamic PL/SQL and SQL
DBMS_TRANSACTION	Transaction management commands
DBMS_UTILITY	Additional utility procedures
UTL_FILE	Provides file I/O