

MP0484.

Bases de datos

UF6. Programación básica de acceso a datos

6.3. Tipos Avanzados de datos

Índice

☰	Objetivos	3
☰	Registros: Record	4
☰	Colecciones limitadas: Varray	8
☰	Colecciones ilimitadas: Table	15
☰	Tablas anidadas simples	17
☰	Tablas anidadas indexadas	25
☰	Operaciones Bulk-Binding	30
☰	Definición y tratamiento	38

Objetivos

Para crear un tipo avanzado de datos, primero definimos el Tipo, y luego creamos una variable de este tipo.



Vamos a estudiar tres tipos avanzados: Registros, Colecciones y Cursores variables. Cada uno con sus particularidades y tratamientos.

Objetivos:

- Entender qué es un tipo avanzado de dato.
- Trabajar con registros.
- Saber las diferencias entre las colecciones.
- Trabajar con colecciones limitadas en elementos: Varray.
- Entender cómo se trabaja con colecciones "ilimitadas".
- Comprender los usos de tablas anidadas indexadas por números y caracteres.
- Ver las diferencias entre cursores estáticos y variables.

Registros: Record

Un registro en un campo compuesto por un conjunto de variables de datos , cada uno con su propio nombre y tipo de datos.

Puedes pensar en un registro como una variable que puede contener una fila de una tabla, o algunas columnas de una fila de una tabla.

Se puede acceder a cada campo de una variable de registro por su nombre, con la siguiente sintaxis:

nombre_variable.nombre_campo.

Hasta este momento hemos visto y trabajado con dos tipos de estructuras %ROWTYPE:

1. Las generadas a partir de una tabla.
2. Las generadas a partir de un cursor.

Ahora vamos a tratar un tercer tipo, que consiste en generar una estructura o registro de los campos que yo considere oportunos, atendiendo al criterio que yo elija.

Tipos de registro

En la figura se muestra un esquema de los tres tipos de registros: de tipo avanzado(**registro**), de fila de tabla(**reg_jobs**) y de cursor(**reg_dep**).

declare

Type registro is record(campo_1 pls_integer, campo_2 varchar2(50));
registro |_____|
 campo_1 campo_2

reg_jobs jobs%rowtype;

reg_jobs |_____| |_____| |_____|
 job_id job_title min_salary max_salary

cursor cur_dep is

select department_id, upper(department_name) as dname from departments;
reg_dep cur_dep%rowtype;
reg_dep |_____| |_____|
 department_id dname

Tipos de registro

La sintaxis para la declaración de un RECORD es:

```
TYPE nombre_tipo IS RECORD (Declaración_campo[,Declaración_campo]...);
```

Una vez definido el tipo, me puedo declarar todas las variables de este tipo como quiera, y todas las variables del mismo tipo tendrán la misma estructura y nombres de campos.

Ejemplo:

Lo vemos con un ejemplo, nos declaramos un tipo registro con dos campos, el apellido y el salario(apel, salario) de la tabla employees, luego me creo una variable de este tipo, y en la cláusula INTO de select recojo el contenido de las dos columnas seleccionadas en mi registro(reg_empl):

```
set serveroutput on
declare
    type tipo_reg_empl is record(apel employees.last_name%type,
                                   salario employees.salary%type);
    reg_empl tipo_reg_empl;

begin
    select last_name, salary
    into reg_empl
    from employees
    where employee_id = 107;

    dbms_output.put_line('apellido : ' || reg_empl.apel);

end;
/
apellido : Lorentz
```

PL/SQL procedure successfully completed.



Siempre que necesite juntar dos o más variables en la misma estructura recurriré a **crear variables de Tipo Registro**.

Dentro de un registro puedo hacer referencia a otra variable de tipo registro.

Vemos un ejemplo.

```
declare
    type tipo_rec_fecha is record(dia decimal(2), mes decimal(2), anio decimal(2));
    type tipo_rec_cita is record(medico varchar2(50), hora dec(2), fecha
        tipo_rec_fecha);

        cita_previa tipo_rec_cita;

begin
    select to_char(sysdate,'dd'), to_char(sysdate,'mm'),to_char(sysdate,'yy')
        into cita_previa.fecha
        from dual;
```

```
cita_previa.medico := '&medico';
cita_previa.hora := &hora;

dbms_output.put('tiene cita con el dtr. ' || cita_previa.medico);
dbms_output.put(' a las ' || cita_previa.hora || ' horas');
dbms_output.put_line(' del dia ' || cita_previa.fecha.dia);

exception
  when others then
    dbms_output.put_line('error en bloque ppal mensaje ' || sqle-
rrm);
end;
/
Enter value for medico: Sanchez
Enter value for hora: 12
tiene cita con el dtr. Sanchez a las 12 horas del dia 27

PL/SQL procedure successfully completed.
```

Nos fijamos en las siguientes consideraciones:

- Del tipo **tipo_rec_fecha(dia, mes anio)**, no hemos creado variable. Se ha creado un campo llamado **fecha** dentro del tipo **tipo_rec_cita**.
- En la select usamos tres expresiones de columna para extraer de la fecha del sistema(**sysdate**) el día , el mes y el año, y situarlo en el campo **fecha** de la variable **cita_previa**.
- El médico y la hora la solicitamos de la consola: '**&medico**' y **&hora**, la variable de entorno **&medico** está entre comillas, porque la variable de destino es varchar2.
- La referencia a una variable de un registro, como ya sabemos, se hace por ejemplo, **cita_previa.medico**; y para hacer referencia una variable de un registro que es a su vez variable de un registro, como se ve en el programa, por ejemplo el día: **cita_previa.fecha.dia**.

Colecciones limitadas: Varray



Las colecciones son campos simples o compuestos(registros), de un mismo tipo de dato, que se repiten en memoria un número determinado de veces, y que se distinguen por la posición que ocupan dentro de la colección.

Si sabes otros lenguajes(java, cobol, python...), no intentes hacer comparaciones entre las colecciones de PL y arrays, tuplas, u otros tipos de datos similares.

Una colección es un grupo de
elementos del mismo tipo.

Cada elemento tiene un único subíndice que determina su posición en la colección. PL/SQL ofrece dos tipos de colecciones:

- **VARRAY:** son limitados y tienen tamaño fijo. Lo estudiamos en este apartado.
- **TABLE:** las tablas son anidadas o indexadas y tienen tamaño variable y numero "ilimitado" de elementos.

Varray

Está pensado para colecciones de datos de un tipo determinado, y en las que conocemos el número de elementos que lo componen.

Al ser un tipo avanzado, primero se declara el tipo, segundo se declara una variable de ese tipo. Y por último en la zona de instrucciones se inicializa el varray y se procesa.

Lo vemos con un ejemplo. vamos a crear un Varray con los 7 días de la semana.

```
set serveroutput on
declare
-- DECLARO EL TIPO
    type varra_dias is varray(7) of varchar2(12);
-- DECLARO UNA VARIABLE DE ESTE TIPO
    dias_semana varra_dias;

begin
--INICIALIZO Y CONSTRUO A TRAVES DEL TIPO LOS 7 ELEMENTOS DE SU TIPO
    dias_semana := varra_dias('lunes','martes','miercoles',
                               'jueves','viernes','sabado','domingo');
exception
    when others then
        dbms_output.put_line('error bloque ppal : ' || sqlerrm);
end;
/
```

Fíjate cómo se construyen los elementos del varray, invocando al tipo y entre paréntesis, definiendo los literales del tipo(días de la semana).

Funciones de Varrays

Las colecciones en general disponen de una serie de funciones, que unas son comunes a todas las colecciones y otras son propias de Varrays.

En la siguiente tabla se describen estas funciones:

Método	Tipo	Descripción
DELETE	Procedure	Elimina elementos de una tabla anidada o tabla indexada.
TRIM	Procedure	Elimina elementos desde el final de un varray o tabla anidada.
EXTEND	Procedure	Agrega elementos al final de un varray o tabla anidada.
EXISTS	Function	Retorna TRUE si y solo si existe el elemento especificado de la colección.
FIRST	Function	Retorna el primer índice de la colección.
LAST	Function	Retorna el último índice de la colección.
COUNT	Function	Retorna el número de elementos en la colección.
LIMIT	Function	Retorna el número máximo de elementos que la colección puede tener.
PRIOR	Function	Retorna el índice que precede al índice especificado.
NEXT	Function	Retorna el índice siguiente al índice especificado.

De esta tabla, hay que considerar que TRIM es propio de Varray y Delete es propio de Table. El resto son funciones compartidas.



Para recorrer un varray se emplea fundamentalmente el bucle de tipo FOR, que es el que mejor se ajusta a las colecciones.

Vamos a usar el ejemplo anterior para entender que proporcionan este tipo de funciones.

```
set serveroutput on
declare
-- DECLARO EL TIPO
    type varra_dias is varray(7) of varchar2(12);
-- DECLARO UNA VARIABLE DE ESTE TIPO
    dias_semana varra_dias;

begin
--INICIALIZO Y CONSTRUYO A TRAVES DEL TIPO LOS 7 ELEMENTOS DE SU TIPO
    dias_semana := varra_dias('lunes','martes','miercoles',
                               'jueves','viernes','sabado','domingo');

-- mostramos por consola lo que nos devuelven las funciones
-- limit: capacidad maxima de varray
    dbms_output.put_line('limite : ' || dias_semana.limit);
-- count: cuantos hay construidos
    dbms_output.put_line('cantidad : ' || dias_semana.count);
-- first: el indice del primero construido
```

```
dbms_output.put_line('primero : ' || dias_semana.first);
-- last: el indice del ultimo construido
    dbms_output.put_line('ultimo : ' || dias_semana.last);
-- recorremos el array
    for i in dias_semana.first .. dias_semana.count loop
        dbms_output.put_line('dia de la semana : ' ||dias_semana(i));
    end loop;
exception
    when others then
        dbms_output.put_line('error bloque ppal : ' || sqlerrm);
end;
/
SQL> @D:\plsql_TED\VARRAY_DEFINICION.sql
limite : 7
cantidad : 7
primero : 1
ultimo : 7
dia de la semana : lunes
dia de la semana : martes
dia de la semana : miercoles
dia de la semana : jueves
dia de la semana : viernes
dia de la semana : sabado
dia de la semana : domingo

PL/SQL procedure successfully completed.
```

Consideraciones:

- 1 El límite y la cantidad coinciden porque el varray está completo (los 7 elementos están construidos).
- 2 El primer elemento de un varray siempre tiene índice 1, por eso .FIRST siempre devuelve 1.
- 3 .COUNT y .LAST en un varray siempre van a coincidir.
- 4 Un Varray definido de n elementos, puede tener sus n elementos construidos, o no, es decir, puede haber menos construidos que la capacidad máximo, e incluso puede estar vacío.
- 5 Un elemento que no está construido, no se puede referenciar, se levanta una excepción, y la ejecución se interrumpe.

Varrays incompletos o vacíos

Vamos a ver a través de un ejemplo un Varray de 5 elementos PLS_INTEGER, en que solo se construyen 4. Y vamos a sacar por consola las funciones más significativas:

```
set serveroutput on
declare
    type varra_pares is varray(5) of pls_integer;
    numeros varra_pares;

begin
    numeros := varra_pares(2,4,6,8);
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);

    for i in 1 .. numeros.count loop
        dbms_output.put_line('elemento : ' || numeros(i));
    end loop;
    dbms_output.put_line('saliendo del for');
exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
SQL> @D:\plsql_TED\varray_incompleto.sql
limite : 5
cantidad : 4
primero : 1
ultimo : 4
elemento : 2
elemento : 4
elemento : 6
elemento : 8
saliendo del for

PL/SQL procedure successfully completed.
```

Consideraciones:

1

.LIMIT y .COUNT no coinciden. El límite son 5 números, y hay 4 construidos.

2

.FIRST siempre 1, .LAST y .COUNT siempre iguales.

3

Si referencio al elemento 5. ¿Qué pasaría? Lo vemos en el siguiente código.

```
set serveroutput on
declare
    type varra_pares is varray(5) of pls_integer;
    numeros varra_pares;

begin
    numeros := varra_pares(2,4,6,8);

    for i in 1 .. numeros.count loop
        dbms_output.put_line('elemento : ' || numeros(i));
    end loop;
-- referencia a un elemento no construido => excepcion
    dbms_output.put_line('elemento 5 ' || numeros(5));
    dbms_output.put_line('saliendo del for');
exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
elemento : 2
elemento : 4
elemento : 6
elemento : 8
error en bloque ppal ORA-06533: Subscript beyond count

PL/SQL procedure successfully completed.
```

Al referenciar a la salida del bucle al elemento 5, se levanta la excepción ORA-06533: *Subscript beyond count*.

Ejemplo:

Para poder usar el elemento 5º, primero lo tengo que construir con .EXTEND, que añade un elemento al final del varray(justo el elemento siguiente a .LAST o .COUNT), siempre que no sobrepasemos el límite(lo que causaría una excepcion), y luego le asigno el valor correspondiente.

Lo vemos en nuestro ejemplo.

```
set serveroutput on
declare
    type varra_pares is varray(5) of pls_integer;
    numeros varra_pares;

begin
    numeros := varra_pares(2,4,6,8);
```

```
-- extiendo el 5º elemento
    numeros.extend;
    numeros(5) := 10;
-- recorro el varray
    for i in 1 .. numeros.count loop
        dbms_output.put_line('elemento : ' || numeros(i));
    end loop;
    dbms_output.put_line('saliendo del for');
exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
@D:\plsql_TED\varray_jugando.sql
elemento : 2
elemento : 4
elemento : 6
elemento : 8
elemento : 10
saliendo del for
```

La función **.TRIM** elimina el último elemento construido, y **.TRIM(n)**, elimina los n últimos construidos, sin pasarme. Vemos el ejemplo, vamos a destruir los dos últimos y luego listamos, nos saldrán los tres primeros.

```
set serveroutput on
declare
    type varra_pares is varray(5) of pls_integer;
    numeros varra_pares;

begin
    numeros := varra_pares(2,4,6,8);

    -- extiendo el 5º elemento
    numeros.extend;
    numeros(5) := 10;
    -- elimino los dos ultimos trim(2)
    numeros.trim(2);
    -- recorro el varray
    for i in 1 .. numeros.count loop
        dbms_output.put_line('elemento : ' || numeros(i));
    end loop;
    dbms_output.put_line('saliendo del for');
exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
@D:\plsql_TED\varray_jugando.sql
elemento : 2
elemento : 4
elemento : 6
saliendo del for
```

PL/SQL procedure successfully completed.

Colecciones ilimitadas: Table

Las Tablas anidadas son colecciones formadas por elementos de un solo tipo de dato, simple o compuesto(Record o %Rowtype), que no tienen límite de elementos: si mostramos la función .LIMIT no devolverá nada.

Normalmente se crean vacías, y se utilizan para almacenar temporalmente en memoria la información procedente de las filas obtenidas por una Select, a través de un cursor.

Existen dos tipos de tablas anidadas:

- **Simples:** el primer elemento construido, a partir de una tabla vacía estará en posición 1.
- **Indexadas:** se pueden indexar de dos formas: por un pls_integer, y entonces disponemos de elementos entre -2 GigaBytes y +2GigaBytes; o por un varchar2(n caracteres), este tipo de indexación es extraño pero veremos su utilidad.

Para que te vaya sonando vamos a poner una definición de cada tipo, y en los siguientes capítulos estudiaremos cada uno.

Ejemplo:

El ejemplo supone que vamos a hacer una select del apellido, salario de los empleados del departamento 30, y los vamos a almacenar en una tabla en memoria para un tratamiento posterior. Como no podemos hacer una tabla de dos campos, primero creamos un registro con estos dos campos y luego creamos tipos de cada opción de tabla.

```
declare
type tipo_reg_empl is record(apel employees.last_name%type,
                               salario employees.salary%type);

type tab_simple is table of tipo_reg_empl;

type tab_index_pls is table of tipo_reg_empl index by pls_integer;

type tab_index_varchar is table of tipo_reg_empl index by varchar2(30);
begin
null;
end;
/
PL/SQL procedure successfully completed.
```

Declaración de cada Tipo de tabla anidada

Las instrucciones del final es para que lo puedas ejecutar y ver que no tiene errores la definición. Fíjate en cómo está creado cada tipo.

A continuación declararíamos variables de cada tipo, y en begin los trataríamos.



Ahora vamos a analizar cada uno.

Tablas anidadas simples

Una tabla anidada simple, se usa normalmente para almacenar temporalmente las filas procedentes de una consulta realizada en una operación cursor.

Estudio de las funciones de Table simples

Pero para entender un poco el funcionamiento de estas tablas, vamos a crear una tabla de números(que son fáciles de manejar) y vamos a mostrar las funciones para ver qué información nos ofrece.

```
set serveroutput on
declare
    type tab1 is table of pls_integer;
    numeros tab1;

begin
    numeros := tab1(11,33,55,66);
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);

    for i in numeros.first .. numeros.last loop
        dbms_output.put_line('elemento : ' || numeros(i));
    end loop;

    dbms_output.put_line('saliendo del for');
exception
```

```
when others then
    dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
limite :
cantidad :4
primero :1
ultimo :4
elemento :11
elemento :33
elemento :55
elemento :66
saliendo del for
```

Consideraciones:

- 1 Aparentemente es como un Varray, pero si te fijas el .LIMIT no devuelve nada: "no hay límite".
- 2 El primer elemento construido es el índice 1.
- 3 De momento, el .COUNT(cuantos hay construidos) y el .LAST(cual es el índice del último construido), es el mismo.
- 4 Podemos destruir el elemento que queramos con .DELETE(no como en Varray que TRIM destruía siempre del ultimo para atrás), dejando huecos entre los elementos. Destruir elementos que previamente se han cargado en memoria, aunque ahora te parezca raro tiene mucha transcendencia en operaciones **BULK BINDING** que veremos en el último apartado de tablas.

Borrado de elementos

El Borrado de elementos de una tabla tiene estas tres posibilidades:

- DELETE: elimina todos los elementos de la colección.
- DELETE(n): elimina el n-ésimo elemento de una tabla anidada o indexada; si este elemento no existe, no hace nada.
- DELETE(m,n): elimina los elementos del rango m..n de una tabla anidad o indexada.
- En los varray, no se puede utilizar DELETE para eliminar elementos en forma individual.

```
cursos.DELETE(2);      -- elimina el elemento 2
cursos.DELETE(7,7);    -- elimina el elemento 7
cursos.DELETE(6,3);    -- no hace nada
cursos.DELETE(3,6);    -- elimina el elemento entre 3 y 6 incluidos
cursos.DELETE;         -- elimina todo los elementos
```

Ejemplo 1:

En el ejemplo que estamos manejando vamos a eliminar el elemento 1. Y nos vamos a fijar en los valores de .FIRST. recuerda que .FIRST nos devuelve el índice **del primer elemento construido.**

```
declare
    type tab1 is table of pls_integer;
    numeros tab1;

begin
    numeros := tab1(11,33,55,66);
    dbms_output.put_line('funciones antes de delete');
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);

    numeros.delete(1);

    dbms_output.put_line('funciones después de delete');
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);
    for i in numeros.first .. numeros.last loop
        dbms_output.put_line('elemento : ' || numeros(i));
    end loop;

    dbms_output.put_line('saliendo del for');

exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
SQL> @D:\plsql_TED\table_simple.sql
funciones antes de delete
```

```
limite :  
cantidad : 4  
primero : 1  
ultimo : 4  
funciones despues de delete  
limite :  
cantidad : 3  
primero : 2  
ultimo : 4  
elemento : 33  
elemento : 55  
elemento : 66  
saliendo del for
```

PL/SQL procedure successfully completed.

Fíjate en los valores devueltos después de destruir el elemento de índice 1.

Consideraciones:

1

.COUNT da 3 porque hay tres construidos, .LAST devuelve 4 porque es el último construido.

2

.FIRST nos ha devuelto 2, porque el primero construido está en el índice 2.

3

Se destruyen los elementos, pero el resto de elementos permanece en su posición original.

4

El bucle nos ha funcionado, porque desde el primero .FIRST al último .LAST, no hay huecos.

Ejemplo 2:

Pero qué pasa cuando destruimos uno del medio (en el ejemplo vamos a destruir el segundo elemento de los 4 del principio), que del primero al último hay huecos, es decir si hacemos referencia a un elemento que no existe, se levanta una excepción, NO_DATA_FOUND (la misma que vimos en otros capítulos pero con distinto significado):

```
set serveroutput on
declare
    type tab1 is table of pls_integer;
    numeros tab1;

begin
    numeros := tab1(11,33,55,66);
    dbms_output.put_line('funciones antes de delete');
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);

    numeros.delete(2);

    dbms_output.put_line('funciones después de delete');
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);
    for i in numeros.first .. numeros.last loop
        dbms_output.put_line('elemento : ' || numeros(i));
    end loop;

    dbms_output.put_line('saliendo del for');

exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
SQL> @D:\plsql_TED\table_simple.sql
funciones antes de delete
limite :
cantidad : 4
primero : 1
ultimo : 4
funciones despues de delete
limite :
cantidad : 3
primero : 1
ultimo : 4
elemento : 11
error en bloque ppal ORA-01403: no data found

PL/SQL procedure successfully completed.
```

Ejemplo 3:

Para leer una tabla que sabemos que tiene huecos y que no de produzcan excepciones debemos preguntar en cada interacción si el elemento referenciado existe o no, a través de la función .EXISTS, fíjate en el ejemplo dentro del bucle FOR.

```
set serveroutput on
declare
    type tab1 is table of pls_integer;
    numeros tab1;

begin
    numeros := tab1(11,33,55,66);
    dbms_output.put_line('funciones antes de delete');
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);

    numeros.delete(2);

    dbms_output.put_line('funciones después de delete');
    dbms_output.put_line('limite : ' || numeros.limit);
    dbms_output.put_line('cantidad : ' || numeros.count);
    dbms_output.put_line('primero : ' || numeros.first);
    dbms_output.put_line('ultimo : ' || numeros.last);

    for i in numeros.first .. numeros.last loop

        IF NUMEROS.EXISTS(i) THEN
            dbms_output.put_line('elemento : ' || numeros(i));
        END IF;

    end loop;

    dbms_output.put_line('saliendo del for');

exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
SQL> @D:\plsql_TED\table_simple.sql
funciones antes de delete
limite :
cantidad :4
primero :1
ultimo :4
funciones despues de delete
limite :
cantidad :3
primero :1
ultimo :4
elemento :11
elemento :55
elemento :66
saliendo del for
```

PL/SQL procedure successfully completed.

```
IF NUMEROS.EXISTS(I) THEN  
    dbms_output.put_line('elemento : ' || numeros(i));  
END IF;
```

Es la clave cuando queremos leer una tabla con huecos.

Cargar tablas simples desde cursos

Hasta ahora lo que hemos hecho es jugar con tablas para comprender su funcionalidad. Ahora vamos a ver un tratamiento típico para almacenar temporalmente información procedente de consultas(select).

El ejemplo sería: queremos consultar el apellido y el salario, de los empleados del departamento 30, y almacenar las filas resultantes en memoria para un posterior tratamiento.

Ejemplo:

El bloque PL/SQL se hace en los siguientes pasos (ver el ejemplo siguiente):

- 1 Definimos el registro con los tres campos.
- 2 Definimos el TYPE de la tabla de ese registro.
- 3 Creamos una variable de este tipo.
- 4 Creamos el cursor para la select correspondiente.
- 5 Ya en Begin, inicializamos la tabla vacía llamando al tipo.
- 6 Invocamos al FOR optimizado del cursor para meter cada fila leída en cada elemento de la tabla.
- 7 Leemos de la tabla simple para comprobar que hemos almacenado el resultado correcto.

```
set serveroutput on
declare
-- 1      type tipo_reg_empl is record(apel employees.last_name%type,
-- 2                      salario employees.salary%type);
-- 3      type tab_empl is table of tipo_reg_empl;
-- 4
-- 5      mi_tab tab_empl;
-- 6      indice pls_integer := 1;
-- 7
-- begin
-- 5      mi_tab := tab_empl();
-- 6
-- 7      for reg_empl in cur_empl loop
--          mi_tab.extend;
--          mi_tab(indice) := reg_empl;
--          indice := indice + 1;
--      end loop;
-- 7
-- 8      for i in mi_tab.first .. mi_tab.last loop
--          dbms_output.put('apellido : ' || mi_tab(i).apel);
--          dbms_output.put_line(' salario : ' || mi_tab(i).salario);
--      end loop;
exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
```

```
/
SQL> @D:\plsql_TED\tab_simple_cursor.sql
apellido : Raphaely salario : 11000
apellido : Khoo salario : 3100
apellido : Baida salario : 2900
apellido : Tobias salario : 2800
apellido : Himuro salario : 2600
apellido : Colmenares salario : 2500
```

PL/SQL procedure successfully completed.

La variable indice me es necesaria para el bucle FOR del cursor.

Puedo igualar el registro del cursor a mi tabla, porque tienen la misma estructura. Si fueran estructuras diferentes habría que hacerlo campo a campo:

- mi_tab(indice).apel := reg_empl.last_name;
- mi_tab(i).salario := reg_empl.salary;

Tablas anidadas indexadas



Las tablas indexadas no hay que inicializarlas, los elementos se van construyendo según vamos referenciando posiciones de memoria a través del índice correspondiente.

Tablas indexadas por enteros

En este tipo de tablas disponemos de posiciones de memoria entre -2gbytes y + 2GBytes, incluyendo la posición cero para introducir datos correspondientes al tipo de datos especificado.

Eso sí, si uso índices alternos, al leer la tabla del primero al último, tengo que preguntar si el elemento apuntado existe (esto lo vimos en tablas simples con huecos).

Ponemos un ejemplo para ver el resultado:

```
SET SERVEROUTPUT ON
DECLARE
  TYPE tTabla2 IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  vTabla2 tTabla2;
BEGIN
  vTabla2(3):= 123;
  vTabla2(73) := 456;
  vTabla2(-5) := 789;
  FOR i IN vTabla2.FIRST .. vTabla2.LAST LOOP
    IF vTabla2.EXISTS(i) THEN
      DBMS_OUTPUT.Put_Line( vTabla2(i) );
    END IF;
  END LOOP;
END;
/
789
123
456
```

PL/SQL procedure successfully completed.

Fíjate que aunque hemos creado los elementos en desorden, al listar la tabla nos han aparecido los contenidos del elemento menor(-5) al mayor(73).

Y es lo que buscamos. Son útiles en los casos en los que queremos usar como índices el contenido de una columna de una tabla(con filas) que sea numérico, para almacenar un contenido ahí, y que al listar la tabla(anidada indexada) la información salga en orden de índice. Por ejemplo, queremos seleccionar el apellido, el salario de los departamentos 10, 20, y 30 ordenados por departamento, para un tratamiento(que no vamos a hacer), pero además queremos saber la suma de los salarios de los empleados por cada año de ingreso, y ordenados por el año.

Para hacer esto tendríamos que hacer dos cursores, para leer primero los empleados ordenados por departamentos, y luego un cursor con una función de grupo, y un group by....

Pues bien, para esto nos sirven las tablas indexadas. Hacemos un solo cursor, y al hacer el For del cursor, vamos sumando los salarios, en una tabla indexada por el año de la fecha de ingreso.

```
set serveroutput on
declare
    type t_index_num is table of employees.salary%type index by pls_integer;
    tabla t_index_num;

    cursor cur_empl is
        select to_number(to_char(hire_date,'yyyy')) as anio,last_name,
        salary,department_id
        from employees
        where department_id in (10,20,30)
        order by department_id;
begin
    for reg_empl in cur_empl loop

        if tabla.exists(reg_empl.anio) then
            tabla(reg_empl.anio) := tabla(reg_empl.anio) + reg_empl.salary;

        else
            tabla(reg_empl.anio) := reg_empl.salary;
        end if;

    end loop;
    dbms_output.put_line('sale carga');
    for i in tabla.first .. tabla.last loop
        if tabla.exists(i) then
```

```
dbms_output.put_line('total anio ' || i || ' es : ' ||
tabla(i));
end if;
end loop;
dbms_output.put_line('fin bloque');
exception
when others then
dbms_output.put_line('error en bloque ppal : ' || sqlerrm);
end;
/
sale carga
total anio 2002 es : 11000
total anio 2003 es : 7500
total anio 2004 es : 13000
total anio 2005 es : 11700
total anio 2006 es : 2600
total anio 2007 es : 2500
fin bloque
```

Procedimiento PL/SQL terminado correctamente.

Para comprobar que la información es correcta ejecutamos la sentencia que estaría en el cursor que nos hemos ahorrado:

```
select to_number(to_char(hire_date,'yyyy')) as anio,sum(salary)
from employees
where department_id in (10,20,30)
group by to_number(to_char(hire_date,'yyyy'))
order by 1;
```

ANIO SUM(SALARY)

```
-----
2002    11000
2003    7500
2004    13000
2005    11700
2006    2600
2007    2500
```

6 rows selected.

Tablas indexadas por varchar2

Indexar una tabla por un campo o expresión de tipo caracteres, tiene la misma connotación que los indexados por números. Disponemos de elementos del tipo de datos que elijamos, con un índice ilimitado de caracteres. podemos referenciar índices con el contenido que queramos, eso si, a la hora de listarlos saldrán en orden ascii de menor a mayor de índice.

Mira el ejemplo siguiente, usamos como índice el literal de provincias españolas, y le asignamos el número de habitantes (los datos son ficticios). Eso sí a la hora de recorrer una tabla de este tipo, verás que es muy peculiar. Lo analizamos detrás del ejemplo:

```
set serveroutput on
declare
    type t_prov is table of pls_integer index by varchar2(15);
    provincias t_prov;
    indice varchar2(15);
begin
    provincias('guadalajara') := 80000;
    provincias('albacete') := 150000;
    provincias('madrid') := 4500000;
    provincias('zaragoza') := 250000;

    indice := provincias.first;
    while indice is not null loop
        dbms_output.put_line(rpad(indice,15) || ' habitantes : '
            || provincias(indice));
        indice := provincias.next(indice);
    end loop;
exception
    when others then
        dbms_output.put_line('error en bloque ppal '|| sqlerrm);
end;
/
albacete    habitantes : 150000
guadalajara habitantes : 80000
madrid      habitantes : 4500000
zaragoza    habitantes : 250000
```

PL/SQL procedure successfully completed.

Nos fijamos en este algoritmo:

1. indice := provincias.first;
2. while indice is not null loop
3. dbms_output.put_line(rpad(indice,15) || ' habitantes : ' || provincias(indice));
4. indice := provincias.next(indice);
5. end loop;

1

No podemos hacer un **for** porque el indice es alfanumérico, por tanto nos tenemos que crear una variable del mismo tipo que el **index by** de la tabla(fíjate en la declare). Al indice le asignamos el **.FIRST (provincias.first)** que nos devuelve el indice más bajo , en este caso Albacete.

2

Hacemos un bucle **while**, mientras el indice no sea null, que es lo que obtenemos en el siguiente al último construido.

3

Mostramos el índice(la función rpad, hace que todas las provincias sean de 15 y así el resultado final queda alineado por provincia, ten en cuenta que la provincia es varchar2, y cada literal tiene distinta longitud) y los habitantes al que apunta este indice.

4

pasamos al siguiente indice en orden ascendente, para estos casos está la función **NEXT(indice := provincias.next(indice))**;

5

Fin de bucle while.

Cuando tengas que leer una tabla indexada por varchar2, este es el algoritmo.

Operaciones Bulk-Binding

Claúsula Bulk Collect

Hasta ahora, siempre que hemos necesitado traspasar las filas de una tabla en una consulta SQL que devuelve más de una fila, a una tabla anidada, tanto simple como indexada, definíamos un cursor y pasábamos cada fila obtenida del cursor a cada elemento de la tabla anidada.

Pues bien desde la versión de Oracle 9i, existe la posibilidad de cargar las filas de una consulta SQL, directamente en una tabla anidada(simple o indexada) , sin necesidad de cursor, con una cláusula añadida a la cláusula INTO, denominada BULK COLLECT.

Recuperamos este ejemplo:

"Queremos consultar el apellido y el salario, de los empleados del departamento 30, y almacenar las filas resultantes en memoria para un posterior tratamiento."

```
set serveroutput on
declare
-- 1
  type tipo_reg_empl is record(apel employees.last_name%type,
-- 2           salario employees.salary%type);
  type tab_empl is table of tipo_reg_empl;
-- 3
  mi_tab tab_empl;
  indice pls_integer := 1;
-- 4
  cursor cur_empl is
    select last_name, salary
    from employees
    where department_id = 30;

begin
-- 5
  mi_tab := tab_empl();
-- 6
```

```
for reg_empl in cur_empl loop
    mi_tab.extend;
    mi_tab(indice) := reg_empl;
    indice := indice + 1;
end loop;
-- 7
for i in mi_tab.first .. mi_tab.last loop
    dbms_output.put('apellido : ' || mi_tab(i).apel);
    dbms_output.put_line(' salario : ' || mi_tab(i).salario);
end loop;
exception
when others then
    dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
```

Así quedaría con BULK COLLECT INTO:

```
set serveroutput on
declare
-- 1
    type tipo_reg_empl is record(apel employees.last_name%type,
                                   salario employees.salary%type);
-- 2
    type tab_empl is table of tipo_reg_empl;
-- 3
    mi_tab tab_empl;
    indice pls_integer := 1;
-- 4 No hay cursor

begin
-- 5 no hace falta inicializar la tabla, te lo hace Bulk collect
--     mi_tab := tab_empl();
-- 6 sustituimos el for de cursor por una select bulk collect into
    select last_name, salary
    bulk collect into mi_tab
    from employees
    where department_id = 30;
-- 7
    for i in mi_tab.first .. mi_tab.last loop
        dbms_output.put('apellido : ' || mi_tab(i).apel);
        dbms_output.put_line(' salario : ' || mi_tab(i).salario);
    end loop;
exception
when others then
    dbms_output.put_line('error en bloque ppal ' || sqlerrm);

end;
/
apellido : Raphaely salario : 11000
apellido : Khoo salario : 3100
```

```
apellido : Baida salario : 2900
apellido : Tobias salario : 2800
apellido : Himuro salario : 2600
apellido : Colmenares salario : 2500
```

```
PL/SQL procedure successfully completed.
```

Quitamos las notas y nos queda:

```
set serveroutput on
declare
    type tipo_reg_empl is record(apel employees.last_name%type,
                                   salario employees.salary%type);
    type tab_empl is table of tipo_reg_empl;
    mi_tab tab_empl;

begin
    select last_name, salary
    bulk collect into mi_tab
    from employees
    where department_id = 30;

    for i in mi_tab.first .. mi_tab.last loop
        dbms_output.put('apellido : ' || mi_tab(i).apel);
        dbms_output.put_line(' salario : ' || mi_tab(i).salario);
    end loop;
exception
    when others then
        dbms_output.put_line('error en bloque ppal ' || sqlerrm);
end;
/
```

Aunque es cómodo y muy potente, no siempre se pueden usar estas instrucciones, y tenemos que recurrir a crear cursosres.

Actualización masiva: sentencia FORALL

Esta sentencia pone en contacto una tabla anidada(simple o indexada) con una sentencia de actualización, o insert, o update o delete, de tal forma que se insertan, modifican o borran tantas filas de la tabla especificada como elementos hay en la tabla anidada.

Por tanto hay una sentencia FORALL distinta, para cada mandato de actualización , Insert, Update o Delete:

```
FORALL i in nombre_tabla.first..nombre_tabla.last
    insert into tabla_SQL
        values nombre_tabla(i);

FORALL i in nombre_tabla.first..nombre_tabla.last
    UPDATE tabla_SQL
        set nombre_columna = nombre_tabla(i).variable, ....
        where condicion(es);

FORALL j IN nombre_tabla.FIRST .. nombre_tabla.LAST
    DELETE FROM tabla_SQL
        WHERE nombre_columna = nombre_tabla(j).variable;
```

Un FORALL por sentencia de actualización

FORALL es una sentencia, y en el momento de la ejecución se transforma internamente(en segundo plano) como un bucle.

De la sentencia se sale por dos motivos:

- O porque se ha terminado de procesar la actualización de todos los elementos de la tabla.
- O porque al actualizar una fila se produce una excepción.

Si se produce una excepción, todas las filas que se hayan actualizado hasta ese momento, si se decide confirmar(COMMIT), se actualizan en la base de datos. Este aspecto para actualizaciones de miles de filas es muy interesante, porque en interactivo un update, delete o insert con subselect son operaciones unarias, es decir o se actualizan todas las filas, o si se produce una excepción, se deshace todas las filas actualizadas.

Vamos a estudiar la sentencia FORAL para insert, en dos circunstancias:

- Desde tablas sin huecos.
- Desde tablas con huecos.

Ejemplo:

Para hacer este estudio, vamos a crear una tabla llamada NEW_EMPLOYEES, sin filas, a partir de la tabla de empleados con las siguientes columnas: employee_id, last_name, job_id, department_id.

```
CREATE TABLE NEW_EMPLOYEES AS  
SELECT EMPLOYEE_ID, LAST_NAME, DEPARTMENT_ID, JOB_ID  
FROM EMPLOYEES  
WHERE 1 = 0;
```

Table NEW_EMPLOYEES creado.

CRETATE TABLE NEW_EMPLOYEES

Y alteramos la tabla, creando la PRIMARY KEY sobre employee_id, y las FOREIGN KEY sobre Deparmetent_id y Job_id sobre sus respectivas tablas departments y jobs.

```
ALTER TABLE NEW_EMPLOYEES ADD PRIMARY KEY(EMPLOYEE_ID);  
ALTER TABLE NEW_EMPLOYEES ADD FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS;  
ALTER TABLE NEW_EMPLOYEES ADD FOREIGN KEY(JOB_ID) REFERENCES JOBS;
```

Table NEW_EMPLOYEES alterado.

Table NEW_EMPLOYEES alterado.

Table NEW_EMPLOYEES alterado.

ALTER sobre NEW_EMPLOYEES

FORALL en Insert con tablas sin huecos

Para entender cómo funciona una actualización con FORALL, vamos a hacer el siguiente ejemplo:

Vamos a leer los empleados de la tabla de employees, que pertenecen a los departamentos 10,20, y 30 , con Bulk Collect, y los vamos a introducir en una tabla anidada de new_employees. Y a continuación, vamos a insertar con FORALL los empleados leídos en la tabla NEW_EMPLOYEES.

```
set serveroutput on
declare
    type tab_newempl is table of new_employees%rowtype;
    tab_new tab_newempl;
begin
    select employee_id, last_name, department_id, job_id
    bulk collect into tab_new
    from employees
    where department_id in (10,20,30);

    forall i in tab_new.first..tab_new.last
        insert into new_employees values tab_new(i);
    commit;
    dbms_output.put_line('filas insertadas : ' || sql%rowcount);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
end;
/
filas insertadas : 9
```

Procedimiento PL/SQL terminado correctamente.

FORALL Insert

FORALL en Insert con tablas con huecos

Para entender cómo funciona una actualización con FORALL, en tablas con huecos, vamos a hacer el siguiente ejemplo:

1. Vamos a leer los empleados de la tabla de employees, que pertenecen a los departamentos 60 y 90 , con Bulk Collect, y los vamos a introducir en una tabla anidada de new_employees.
2. Vamos a destruir de la tabla anidada los empleados con numero par de código.
3. Y a continuación, vamos a insertar con FORALL los empleados leídos en la tabla NEW_EMPLOYEES.

```
set serveroutput on
declare
    type tab_newempl is table of new_employees%rowtype;
    tab_new tab_newempl;
begin
    select employee_id, last_name, department_id, job_id
    bulk collect into tab_new
    from employees
    where department_id in (60,90);
    for i in tab_new.first .. tab_new.last loop
        if mod(i,2) = 0 then
            tab_new.delete(i);
        end if;
    end loop;
    forall i in tab_new.first..tab_new.last
        insert into new_employees values tab_new(i);

    dbms_output.put_line('filas insertadas sin errores : ' || sql%row-
count);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('filas insertadas : ' || sql%rowcount);

end;
/
ORA-22160: element at index [2] does not exist
filas insertadas : 1

Procedimiento PL/SQL terminado correctamente.
```

FORALL Insert con huecos: Excepción

Fíjate en la salida: se ha producido la excepción porque la tabla tiene un hueco, es decir, el elemento 2 se ha destruido.

Y el FORALL internamente recorre la tabla del primero al último. Pero recuerda que eso solo se puede hacer para tablas sin huecos.



Para este tipo de operaciones, forall, está la utilidad de tener huecos en las tablas.

Para que el FORALL funcione, sustituye en el código el FORALL por este:

```
FORALL I IN INDICES OF TAB_NEW
    INSERT INTO NEW_EMPLOYEES VALUES TAB_NEW(I);
```

filas insertadas sin errores :4

Procedimiento PL/SQL terminado correctamente.

FORALL tablas con huecos

Al poner este **FORALL i IN INDICES OF TAB_NEW...**, el bucle interno en ejecución, se salta los que están destruidos. Y verás que solo están dados de alta los empleados con código de empleado impar:

- 103
- 105
- 107
- 101

El FORALL es verdaderamente **útil, en operaciones masivas de actualización**, de miles y miles de filas.

Definición y tratamiento

Un cursor variable es un tipo avanzado de dato, por tanto, primero se define el Tipo, y luego una variable de este tipo.

Y posee las siguientes características:

- Al ser una variable, ésta, a diferencia de un cursor estático, se puede pasar como parámetro a una función, o un procedimiento.
- La consulta asociada al cursor variable, no la decido en tiempo de **DECLARE**, la decido en tiempo de ejecución(**BEGIN**).
- Con un cursor variable puedo abrir y cerrar consultas distintas, a tablas distintas con condiciones distintas; a diferencia de un cursor estático que siempre apunta a la misma consulta con las mismas tablas y las mismas condiciones.

Declaración de cursos variables

Para crear cursos variables, se deben de seguir dos etapas.

1. En la primera se define un tipo REF CURSOR, entonces se declaran los cursos variables de ese tipo.
2. En la segunda se crea una variable de este tipo.

Se pueden declarar dos tipos de REF CURSOR, de referencias a cursor:

DECLARE

-- paso 1, definición del tipo

```
TYPE CUR_GENERICO IS REF CURSOR;
```

```
TYPE CUR_ESPECIFICO IS REF CURSOR RETURN tipo_de_dato;
```

-- paso 2 declaración de variables de este tipo

```
cur_1 cur_generico;
```

```
cur_2 cur_especifico;
```

Tipos de cursor variable

1

Tipo genérico: la **SELECT** asociada a cualquier consulta que se haga en tiempo de ejecución, puede contemplar cualquier conjunto de columnas o expresiones de columna.

2

Tipo específico: la **SELECT** que se haga en tiempo de ejecución tiene que seguir el patrón marcado por el tipo de dato que "especifica" el tipo definido, se dice que el cursor "*devuelve*" columnas o expresiones de columna de este tipo de dato. Tipos de datos devueltos: tipos de oracle, más los propios de PL/SQL, más %TYPE, más %ROWTYPE, más RECORD.

Ejecución de cursosres variables

Los cursosres variables, a diferencia de los cursosres estáticos, **NO se puede usar el bucle FOR optimizado**.

La única forma de tratarles es a través de las sentencias:

- OPEN-FOR.
- FETCH.
- CLOSE.

Primero se utiliza OPEN para abrir el cursor variable, y es aquí donde se decide la sentencia SELECT, a través de la cláusula FOR de la sentencia OPEN. A continuación, se realiza una recuperación (FETCH) de las filas resultantes del conjunto una a una. Cuando todas las filas se han procesado es necesario cerrar (CLOSE) el cursor variable.

Sentencia OPEN

La sentencia OPEN-FOR asocia un cursor variable con una consulta multi-fila, ejecuta la consulta, e identifica el conjunto resultante.

La sintaxis es la siguiente:

```
OPEN nombre_cursor_variable  
FOR sentencia_select .....;
```

La sentencia select dependerá del TYPE generado en la declare:

- **Tipo genérico:** cualquier consulta a cualquier columna o columnas, de cualquier tabla o tablas.
- **Tipo específico:** la estructura de las columnas o expresiones de columna de la Select, se tiene que ajustar al tipo devuelto por el cursor., eso sí, de cualquier tabla o join de tablas.

Vemos la apertura en este código:

```
set serveroutput on  
declare  
    type rec_2v2 is record(campo1 varchar2(100), campo2 varcahr2(100));  
    type cur_gnrco is ref cursor;  
    type cur_espc0 is ref cursor return rec_2v2;  
  
    cur1 cur_gnrco;  
    cur2 cur_espc0;  
begin  
    -- cur1 la select la hago de lo que quiero  
    open cur1 for select * from jobs;  
    -- cur2 la select me ajusto a consultas de 2 varchar2 que es lo que marca  
    el tipo  
    open cur2 for select last_name, email from employees where department_id = 30;  
  
    exception  
        when others then  
            dbms_output.put_line('error en bloque ppal : ' || sqlerrm);  
  
end;  
/
```

Open cursor

Sentencia FETCH y CLOSE

Una vez abierto el cursor el tratamiento es el mismo que para cursos estáticos. Nos remitimos a lo escrito en el apartado 6.2 de este contenido.

Resolvemos el ejemplo anterior. Fíjate en la definición de un registro(rec_2v2) formado por dos campos de dos varchar2, que luego le usamos en el into del Fetch del cursor2.

```
set serveroutput on
declare
    type rec_2v2 is record(campo1 varchar2(100), campo2 varchar2(100));
    type cur_gnrco is ref cursor;
    type cur_espco is ref cursor return rec_2v2;

    cur1 cur_gnrco;
    cur2 cur_espco;
    f_jobs jobs%rowtype;
    f_empl rec_2v2;
begin
    -- cur1 la select la hago de lo que quiero
    open cur1 for select * from jobs;
    -- cur2 la select me ajusto a consultas de 2 varchar2 que es lo que marca el tipo
    open cur2 for select last_name, email from employees where department_id = 30;

    loop
        fetch cur1 into f_jobs;
        exit when cur1%NOTFOUND;
        dbms_output.put_line('trabajo : ' || f_jobs.job_title);
    end loop;
    close cur1;

    loop
        fetch cur2 into f_empl;
        exit when cur2%NOTFOUND;
        dbms_output.put_line('apellido : ' || f_empl.campo1);
    end loop;
    close cur2;

exception
    when others then
        dbms_output.put_line('error en bloque ppal : ' || sqlerrm);

end;
/
```

Utilidad

Sobre todo son útiles para trabajar con Procedimientos almacenados, para pasar el cursor como tipo de dato por parámetro a bloques que solicitan leer de tablas. Lo veremos en la Unidad 7.2.



PROEDUCA