



BASES DE DATOS AVANZADAS

Práctica 3, sesión 1

Aspectos Objeto-Relacionales con ORACLE 10g

Univ. Cantabria – Fac. de Ciencias
Francisco Ruiz



Objetivos

- Conocer el modelo de objetos utilizado por ORACLE 10g.
- Aprender a trabajar con los aspectos objetuales de dicho gestor de bases de datos objeto-relacionales:
 - Usar textos grandes, colecciones y tipos objeto.
 - Realizar operaciones de manipulación con tipos implicados.



Agradecimientos

- Este material ha sido preparado con la colaboración de:
 - Coral Calero (Univ. de Castilla-La Mancha)
 - Belen Vela (Univ. Rey Juan Carlos)



Contenido

- Modelo de Objetos.
- Objetos Binarios Grandes.
 - Textos CLOB.
- Colecciones.
 - Vectores.
 - Tablas Indexadas.
 - Tablas Anidadas.
 - Manipulación.
 - Colecciones Multinivel.
- Registros.
 - Creación.
 - Uso.
- Pseudocolumnas.
 - Secuencias.
 - Rowid y Level.
- Tipos Objeto.
 - Creación.
- Cuerpo.
- Modificación.
- Eliminación.
- Objetos Columna.
- Objetos Fila.
- Herencia.
- Referencias.
- Tipos dentro de Tipos.
- Manipulación de Tipos.
 - Inserciones.
 - Actualizaciones.
 - Eliminaciones.
 - Consultas con Objetos Columna.
 - Consultas con Objetos Fila.
 - Consultas con Referencias.
- Vistas Objeto.



Bibliografía

- Básica
 - ORACLE 10g PL/SQL. User's Guide and Reference.
 - 10g Release 1 (10.1), December 2003.
 - Caps. 5 y 12.
- Complementaria
 - ORACLE 10g PL/SQL. User's Guide and Reference.
 - 10g Release 1 (10.1), December 2003.
 - Caps. 3 y 6.
 - Pérez, C. (2008): ORACLE PL/SQL. Ra-Ma.
 - Caps. 4, 6, 7 y 12.



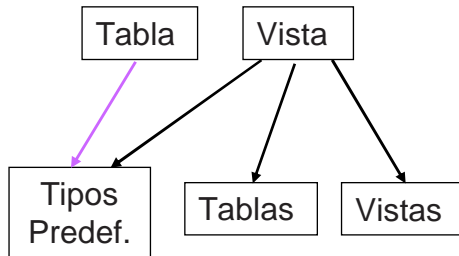
Modelo de Objetos

- **ORACLE 10g** utiliza un **modelo de objetos** con algunas diferencias respecto de **SQL:2003**:

SQL:2003	ORACLE 10g
Tipo Estructurado (UDT)	Tipo Objeto
Tabla Tipada (<i>CREATE TABLE .. OF ..</i>)	Tabla de Objetos
Subtabla (herencia entre tablas)	No soportada directamente. Se debe hacer mediante Vistas de Objetos creadas sobre la tabla base que es supertabla.
Vector (<i>Array</i>)	Varray
Multiconjunto (<i>Multiset</i>)	Tabla Anidada (Nested Table)
Tipo Distinto (<i>Distinct</i>)	Subtipo (Subtype)

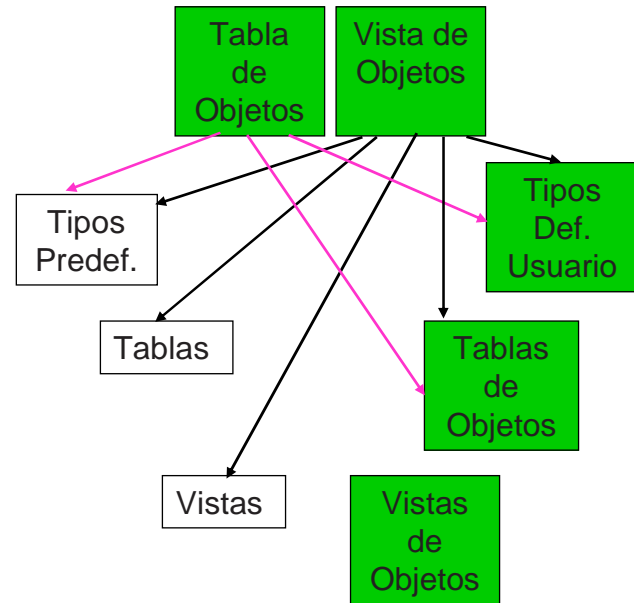


Relacional



Ansteiy (1998)

Objeto-Relacional



- Los **LOBs (Large Objects)** son tipos de datos con gran capacidad de almacenamiento (hasta **4 Gb**).
- Un LOB consta de dos partes:
 - **Localizador** (Locator): Puntero que especifica la localización del contenido.
 - **Contenido** (Content): Los datos de tipo carácter o byte almacenados en el LOB.



Objetos Binarios Grandes

- **Tipos LOB:**

- **BFILE** (*Binary File*): Localizador de un fichero de sistema operativo externo a la base de datos.
- **BLOB** (*Binary Large Object*): Objetos binarios largos (video, audio, etc.).
- **CLOB** (*Character Large Object*): Texto largo formado por caracteres estándares.
- **NCLOB**: (*National Character Large Object*): Texto largo formado por caracteres de varios bytes (vocabularios nacionales especiales: árabe, japonés, chino, etc.).

- En el tipo BFILE el contenido se almacena fuera de la base de datos (el localizador apunta al fichero externo).
- En los demás tipos el contenido se almacena en la propia base de datos, dentro de la misma tabla (si tiene menos de 4kb de tamaño) o fuera de ella.



Objetos Binarios Grandes

- **Creación e Inicialización de LOB:**

- Creación:

```
CREATE TABLE clob_ej (  
    id          INTEGER PRIMARY KEY,  
    clob_col    CLOB NOT NULL,  
    blob_col    BLOB  
);
```

- Inicialización:

```
INSERT INTO clob_ej (id, clob_col, blob_col)  
VALUES (1, EMPTY_CLOB(), EMPTY_BLOB())  
);
```



Objetos Binarios Grandes

Ejemplos:

```

CREATE TABLE Multimedia_Table (
  Clip_ID      NUMBER NOT NULL,
  Story        CLOB default EMPTY_CLOB(),
  FLSub        NCLOB default EMPTY_CLOB(),
  Photo        BFILE default NULL,
  Frame        BLOB default EMPTY_BLOB(),
  Sound        BLOB default EMPTY_BLOB(),
  Voiced_ref   REF Voiced_typ,
  Music        BFILE default NULL,
  Comments     LONG )
LOB(Frame) STORE AS (TABLESPACE lobtbs1
  ...
  STORAGE (
  INITIAL 100M
  NEXT 100M
  MAXEXTENTS UNLIMITED
  PCTINCREASE 0) );

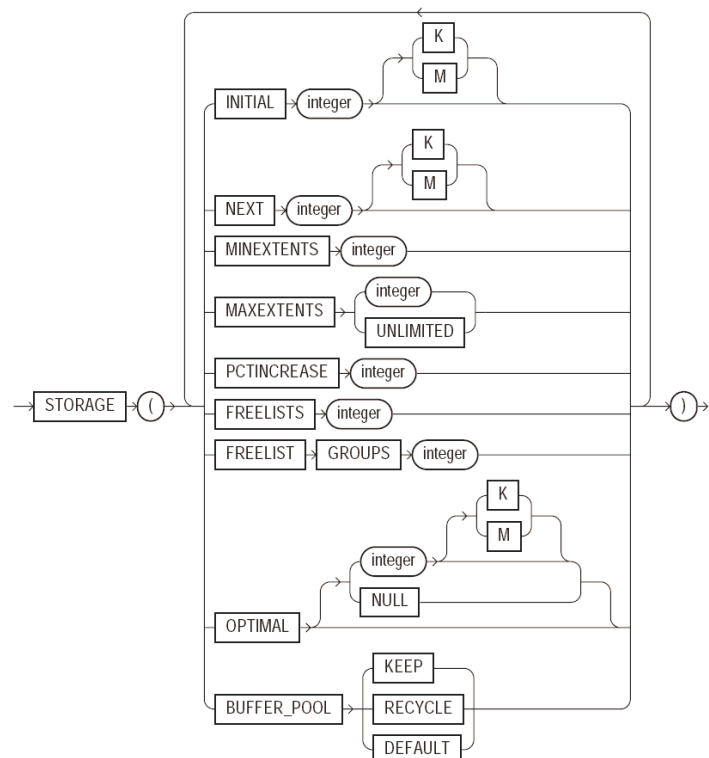
```



Objetos Binarios Grandes

Ejemplos:

- El ejemplo anterior incluye una **cláusula STORAGE** que sirve para indicar el modo de almacenar un objeto de la base de datos.





Objetos Binarios Grandes

- **Añadiendo Contenido:**

- Añadir:

```
UPDATE clob_ej
  SET clob_col = 'En un lugar de La Mancha ...'
  WHERE id = 1;
```

```
UPDATE clob_ej
  SET clob_col = 'Erase una vez en la vida de ...'
  WHERE id = 2;
```

- Consultar:

```
SELECT id, clob_col FROM clob_ej;
```



Objetos Binarios Grandes – Textos CLOB

- Existen dos **funciones SQL** para **manejo de CLOB**:

- **EMPTY_CLOB()**: inicializa un objeto vacío.

```
UPDATE SET articulo = EMPTY_CLOB();
```

- **TO_CLOB(<char>)**: convierte un NCLOB, CHAR, VARCHAR u otro tipo string a CLOB.

```
UPDATE periodicos SET portada = TO_CLOB(txt_inicial);
```

- También pueden aplicarse diversos operadores de manejo de strings, como **concatenación** (||).



Objetos Binarios Grandes – Textos CLOB

- Además, en **PL/SQL** existe el **paquete DBMS_LOB** que provee diversas rutinas:
 - **READ(<lob>,<can>,<pos>,<var>);**
 - Leer una cantidad 'can' de bytes comenzando en 'pos' y guardarlo en la variable 'var'.
 - **WRITE(<lob>,<can>,<pos>,<var>);**
 - Escribir una cantidad 'can' de bytes comenzando en 'pos' leyendo de la variable 'var'.
 - **APPEND(<ori>,<des>);**
 - Añadir el contenido del CLOB 'ori' al CLOB 'des'.
 - **COPY(<des>,<ori>,<cant>,<pos_des>,<pos_ori>)**
 - Copia des el CLOB 'ori' al CLOB 'des' una cantidad 'can' de bytes tomando desde la posición 'pos_ori' y pegando en la posición 'pos_des'.
 - **INSTR(<lob>,<pat>,<pos>,<n>);**
 - Busca el patrón de texto 'pat' a partir de la posición 'pos' y devuelve la posición en que se encuentra la ocurrencia n-sima.



Objetos Binarios Grandes – Textos CLOB

- Además, en **PL/SQL** existe el **paquete DBMS_LOB** que provee diversas rutinas:
 - **CLOSE.** Cerrar.
 - **COMPARE.** Comparar dos CLOB o partes.
 - **ERASE.** Borrar todo o parte.
 - **GETLENGTH.** Obtener longitud.
 - **OPEN.** Abrir.
 - **SUBSTR.** Leer parte.
 - **WRITEAPPEND.** Añadir al final desde una variable.



Colecciones

- Una **colección** es un grupo ordenado de elementos del mismo tipo.
 - Las colecciones tienen una única dimensión.
 - Es factible modelar **colecciones multidimensionales** creando colecciones cuyos elementos son a su vez colecciones.
 - Para usarlas se deben definir uno o varios tipos PL/SQL y entonces definir variables con dichos tipos.
 - Se pueden definir tipos colección en un procedimiento, función o paquete.
 - Los elementos de una colección pueden ser tipos definidos por el usuario.



Colecciones

- PL/SQL permite tres **clases de colecciones**:
 - **Vectores** (*Varray*)
 - Número de fijo de elementos identificados por un subíndice secuencial.
 - Pueden ser almacenados en tablas base.
 - **Tablas Indexadas** (*Index-by Table*).
 - También llamados Vectores Asociativos (*Associative Array*).
 - Similares a tablas hash.
 - Permiten buscar elementos usando claves de búsqueda.
 - **Tablas Anidadas** (*Nested Table*)
 - Parecidas a los Multiconjuntos de SQL.
 - Cada elemento se refiere por un índice secuencial.
 - Pueden almacenarse en tablas base y ser manipuladas con órdenes SQL.



Colecciones

- **Creación de Colecciones:**

- Primero se define el tipo colección y luego se declaran variables de ese tipo.
- Los tipos colección se pueden definir en la **parte declarativa** de un bloque, rutina o paquete PL/SQL:

```
TYPE ...
```

- La colección se instancian al entrar en el bloque o rutina y desaparecen al salir de el.
- En un paquete se instancian al referirlo la primera vez y desaparecen al acabar la sesión.
- También se pueden crear en SQL interactivo:

```
CREATE TYPE ...
```



Colecciones - Vectores

- Los elementos en un **Varray** se almacenan comenzando en el índice 1 hasta la longitud máxima declarada en el tipo.

```
TYPE <nombre> IS {VARRAY | VARYING ARRAY}  
(<tamaño>) OF <tipo> [NOT NULL];
```

- Tipo de los elementos:

- No puede ser BOOLEAN, NCHAR, NCLOB, NVARCHAR(n), REF CURSOR, TABLE u otro VARRAY.
- Se puede especificar utilizando %TYPE y %ROWTYPE:

```
TYPE lcodigos IS VARRAY (52) OF provincias.cp%TYPE;
```



Colecciones - Vectores

- **Ejemplo de Varray en SQL:**

```
CREATE OR REPLACE TYPE tipotelefono
AS VARRAY(3) OF VARCHAR2(10);
```

```
CREATE TABLE empleado (
dni          NUMBER,
nombre      VARCHAR2(30),
telefonos   tipotelefono);
```

```
INSERT INTO empleado VALUES ('9876543', 'Pepe',
TipoTelefono('914445566', '606445566', '934445566'));
```



Colecciones - Vectores

- **Ejemplo de Varray en PL/SQL:**

- Los Varray se inicializan usando un constructor.

```
CREATE PROCEDURE ...
```

```
DECLARE
TYPE tipo_numeros IS VARRAY(20) OF
NUMBER(3);
nros_uno tipo_numeros;
nros_dos tipo_numeros :=
tipo_numeros(1,2);
nros_tres tipo_numeros :=
tipo_numeros(NULL);
```

```
...
```

```
BEGIN
IF nros_uno IS NULL THEN
DBMS_OUTPUT.PUT_LINE(' nros_uno
es NULL');
END IF;
IF nros_tres IS NULL THEN
DBMS_OUTPUT.PUT_LINE(' nros_tres
es NULL');
END IF;
IF nros_tres(1) IS NULL THEN
DBMS_OUTPUT.PUT_LINE('
nros_tres(1) es NULL');
END IF;
END;
```



Colecciones - Vectores

- El **tamaño inicial** de un **VARRAY** se establece mediante el número de elementos indicados en el constructor usado para declararlo.

DECLARE

```
TYPE t_cadena IS VARRAY(5) OF VARCHAR2(10);
```

```
v_lista t_cadena := ('Paco', 'Pepe', 'Luis'); -- tamaño de 3
```

BEGIN

```
v_lista(2) := 'Lola';
```

```
v_lista(4) := 'Está mal'; -- dá error
```

END;

/

- El tamaño puede aumentarse utilizando la función **EXTEND**, pero nunca con mayor dimensión que la definida en la declaración del tipo.
 - v_lista se puede aumentar a 4 o 5 elementos.



Colecciones - Vectores

- Pueden almacenarse **en columnas de tablas**.

```
CREATE OR REPLACE TYPE lista_libros AS VARRAY(10) OF  
inf_libro;
```

```
CREATE TABLE ejemplo (  
id      number,  
libros  inf_libro);
```

- Para modificar un vector almacenado:
 - Seleccionarlo en una variable PL/SQL.
 - Modificar la variable.
 - Volver a almacenar en la tabla.



Colecciones – Tablas Indexadas

- Una **tabla indexada** tiene dos componentes:
 - Una clave de tipo numérico o carácter, que permite indexar la tabla.
 - Una columna de escalares o registros que contiene los elementos de la tabla.
- Puede incrementar su tamaño dinámicamente (con **EXTEND**).
- **Creación:**

```
TYPE <nombre> IS TABLE OF <tipo> [NOT NULL]
INDEX BY [PLS_INTEGER | BINARY_INTEGER |
VARCHAR2(<longitud>)];
```



Colecciones – Tablas Indexadas

- **Ejemplo**

```
DECLARE
    TYPE ttindex IS TABLE OF VARCHAR2(32) INDEX BY
        BINARY_INTEGER;
    v          ttindex;
BEGIN
    v('Canada') := 'NorteAmerica';
    v('Grecia') := 'Europa';
END;
```



Colecciones – Tablas Anidadas

- Las **tablas anidadas** son parecidas a las tablas indexadas.
- Equivalen a los multiconjuntos de SQL estándar.

`TYPE <nombre> IS TABLE OF <tipo> [NOT NULL];`

- El tipo de los elementos no puede ser:
 - En PL/SQL: REF CURSOR.
 - En SQL (CREATE TYPE): BINARY_INTEGER, PLS_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, POSITIVE, POSITIVEN, REF CURSOR, SIGNTYPE, STRING.



Colecciones – Tablas Anidadas

- **Ejemplo en PL/SQL:**

```
DECLARE
  TYPE ttanidada IS TABLE OF VARCHAR2(20);
  v ttanidada ;
BEGIN
  v := ttanidada('Hola','Aidos','Chao','Saludos');
END;
```
- **Ejemplo en SQL:**

```
CREATE TYPE listaTif AS TABLE OF VARCHAR2(10) -- definición
/
CREATE TYPE estudiante AS OBJECT (
  id          INTEGER(4),
  nombre     VARCHAR2(25),
  direc      VARCHAR2(35),
  estado     CHAR(2),
  telefonos  listaTif); -- tabla anidada como atributo multivaluado
/
```



Colecciones – Tablas Anidadas

- Las tablas anidadas se **inicializan** de forma similar a los vectores:

```
DECLARE
  TYPE colores IS TABLE OF VARCHAR2(16);
  arcoiris      colores;
  miscolores    colores := colores('añil','negro','rojo');
BEGIN
  arcoiris :=
  colores('Rojo','Naranja','Amarillo','Verde','Azul','Índigo','Violeta');
END;
/
```

- Se pueden hacer asignaciones de elementos:

```
<nombre_colección>(<índice>) := <expresión>;
colores(1) := 'rosa';
```



Colecciones – Manipulación

- Métodos** para manejar **colecciones**:

- Son funciones o procedimientos preconstruidos para facilitar el uso de colecciones.
- Se invocan mediante **<colección>.<método>**.
- NO pueden ser llamados desde sentencias SQL.
- Son:

EXISTS	COUNT
LIMIT	FIRST and LAST
PRIOR and NEXT	EXTEND
TRIM	DELETE
- EXTEND y TRIM no pueden usarse con tablas indexadas.
- EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR y NEXT son funciones.
- EXTEND, TRIM y DELETE son procedimientos.
- EXISTS, PRIOR, NEXT, TRIM, EXTEND y DELETE tienen como parámetro el subíndice del elemento afectado.



Colecciones – Manipulación

- **Métodos** para manejar **colecciones**:
 - **EXISTS(i)**. Saber si en un cierto índice hay almacenado un valor.
 - **LIMIT**. Número máximo de elementos de un vector.
 - **COUNT**. Número de elementos de la colección.
 - **FIRST** y **LAST**. Devuelven el menor y mayor índice de la colección o NULL si está vacía.
 - **PRIOR(n)** y **NEXT(n)**. Devuelven el índice anterior o posterior a n de la colección o NULL si no se encuentra.



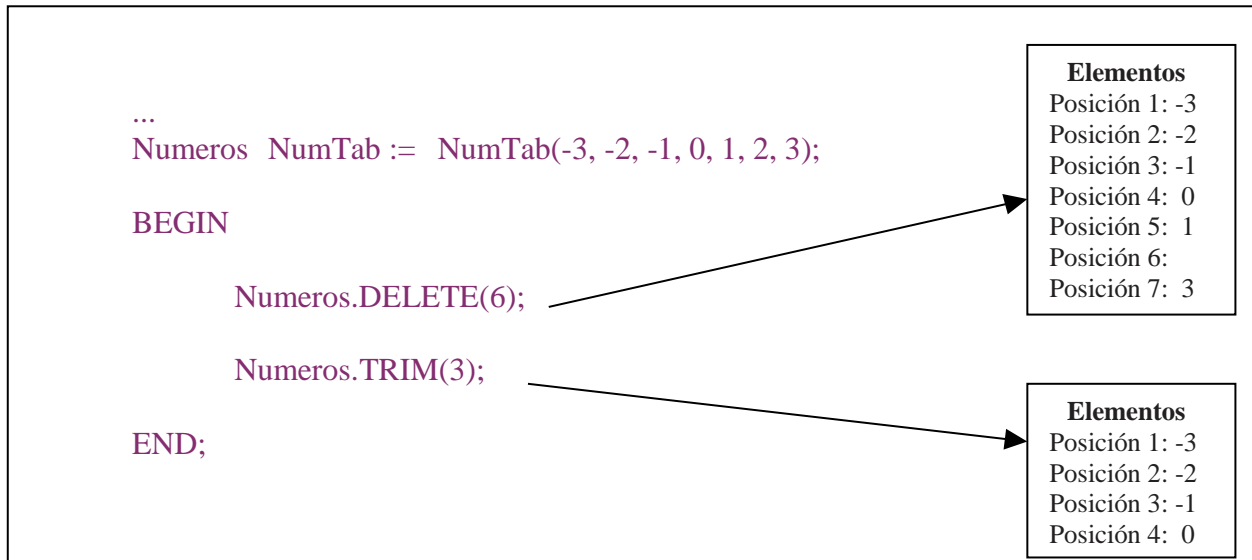
Colecciones – Manipulación

- **Métodos** para manejar **colecciones**:
 - **EXTEND**. Incrementar el tamaño de una tabla anidada o vector.
 - **EXTEND** => Añadir un elemento nulo al final.
 - **EXTEND(n)** => Añadir n elementos nulos al final.
 - **EXTEND(n,i)** => Añadir n copias del elemento i-ésimo al final.
 - **TRIM**. Reducir el tamaño.
 - **TRIM** => Eliminar el último elemento.
 - **TRIM(n)** => Eliminar los n últimos elementos.
 - **DELETE**. Borrar elementos.
 - **DELETE** => Elimina todos los elementos.
 - **DELETE(n)** => Elimina el elemento n-ésimo de una tabla indexada (con clave numérica) o una tabla anidada.
 - **DELETE(m,n)** => Elimina los elementos entre el m-ésimo y el n-ésimo.



Colecciones – Manipulación

- Diferencias entre DELETE y TRIM.



Colecciones – Multinivel

- Se pueden crear colecciones cuyos elementos son colecciones:
 - Tabla anidada formada por vectores,
 - Vector formado por vectores (matriz),
 - Vector formado por tablas anidadas,
 - Tabla anidada formada por tablas anidadas, etc.

```

DECLARE
    TYPE t1 IS VARRAY(10) OF INTEGER;
    TYPE nt1 IS VARRAY(10) OF t1; -- vector de vectores
    va t1 := T1(2,3,5);
    -- inicialización del vector multidimensional
    nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
    i INTEGER;
BEGIN
    -- acceso
    i := nva(2)(3); -- devuelve el valor 73

```



Colecciones – Multinivel

- Ejemplo con **Tablas Anidadas multinivel**.

DECLARE

```
TYPE tb1 IS TABLE OF VARCHAR2(20); -- tipo tabla anidada de strings
```

```
TYPE ntb1 IS TABLE OF tb1; -- tipo tabla de elementos tipo tabla
```

```
TYPE tv1 IS VARRAY(10) OF INTEGER; -- tipo vector
```

```
TYPE ntb2 IS TABLE OF tv1; -- tipo tabla de elementos tipo vector
```

```
vtb1 tb1 := tb1('UNO', 'TRES'); -- tabla de elementos string
```

```
vntb1 ntb1 := ntb1(vtb1); -- tabla de elementos tabla
```

```
vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3)); -- tabla de elementos vector
```

BEGIN

```
vntb1.EXTEND;
```

```
vntb1(2) := vntb1(1);
```

```
vntb1.DELETE(1); -- borra el primer elemento
```

```
-- borra el primer string de la segunda tabla que es elemento de la tabla anidada
```

```
vntb1 vntb1(2).DELETE(1);
```

END;

/



Registros

- Un **Registro** es un grupo de items de datos relacionados almacenados en campos.
 - Cada campo tiene un nombre y un tipo de dato.
 - Los campos pueden ser inicializados y pueden ser definidos como NOT NULL.
- Los registros pueden estar anidados, unos dentro de otros.
 - Con **%ROWTYPE** se puede declarar un registro cuyos campos son los mismos y del mismo tipos que las columnas de una tabla.



Registros - Creación

- Para **crear registros**
 - Definir un tipo RECORD
 - Declarar variables de dicho tipo.
- Las definiciones de tipo **RECORD** se pueden hacer en:
 - La parte declarativa de un bloque, una rutina o un paquete PL/SQL.
 - En SQL (con CREATE TYPE ...)



Registros - Creación

- **Declaración**

```
TYPE <nombre> IS RECORD (<campo> [, <campo> ...]);  
  
<campo> := <nombre> <tipo> [NOT NULL] [{:= | DEFAULT}  
  <expresión>]  
  
<tipo> := {<tipo_dato> | <variable%TYPE> |  
  <tabla>.<columna>%TYPE | <tabla>%ROWTYPE}
```
- **Ejemplo:**

```
DECLARE  
TYPE treg IS RECORD (  
  campo1 VARCHAR2(16) NOT NULL, campo2 NUMBER := 0, campo3  
  DATE);  
vr1 treg;
```



Registros - Creación

- **Ejemplo de Declaración con %ROWTYPE**

```
CREATE TABLE empleado (
```

```
    id number,
```

```
    nombre char(10),
```

```
    apellido char(20),
```

```
    dirección char(30));
```

- Declaración de una variable de tipo registro como:

```
DECLARE
```

```
    reg_emp empleado%ROWTYPE;
```



Registros - Uso

- **Asignando valores**

```
DECLARE
```

```
    TYPE treg IS RECORD (
```

```
        campo1 VARCHAR2(16) NOT NULL, campo2 NUMBER := 0,  
        campo3 DATE);
```

```
    vr1 treg;
```

```
    vr2 treg;
```

```
BEGIN
```

```
    -- asignar valores a campos de vr1
```

```
    vr1.campo1 := 'Estatua ecuestre'; vr1.campo2 := 100;
```

```
    -- asignar a un registro el valor de otro (son del mismo tipo)
```

```
    vr2 := vr1;
```

```
    ...
```

```
END;
```

```
/
```



Registros - Uso

- **Asignando valores con una Consulta**

- También se pueden asignar valores de un registro completo mediante una SELECT que extraería los datos de la BD y los almacena en el registro.

```
SELECT nombre, apellido, carrera
```

```
INTO V_Registro
```

```
FROM Estudiantes
```

```
WHERE ID_Estudiante=1000;
```

V_Registro es una variable de tipo T_Registro:

```
TYPE T_Registro IS RECORD(  
Nombres Estudiantes.nombre%TYPE,  
Apellidos Estudiantes.apellido%TYPE,  
Carrera Estudiantes.carrera%TYPE);
```



Registros - Uso

- **Insertando y Modificando datos de una tabla**

```
DECLARE
```

```
-- deptno, dnombre y loc son las columnas de la tabla dept  
dept_info dept%ROWTYPE;
```

```
BEGIN
```

```
dept_info.deptno := 70;
```

```
dept_info.dnombre := 'PERSONAL';
```

```
dept_info.loc := 'Santillana';
```

```
INSERT INTO dept VALUES dept_info;
```

```
dept_info.deptno := 30;
```

```
dept_info.dname := 'MARKETING';
```

```
dept_info.loc := 'Almagro';
```

```
UPDATE dept SET ROW = dept_info WHERE deptno = 30;
```

```
END;
```

```
/
```



- **Consultando colecciones de registros**

- La cláusula BULK COLLECT en una consulta permite recuperar un conjunto de filas en una colección de registros.

DECLARE

```
TYPE EmpleadoSet IS TABLE OF empleados%ROWTYPE;  
ve EmpleadoSet;
```

BEGIN

```
-- recuperamos y guardamos en la colección  
SELECT * BULK COLLECT INTO ve FROM empleados  
  WHERE salario < 2500 ORDER BY salario DESC;  
-- procesamiento de los datos de la colección de registros  
dbms_output.put_line(ve.COUNT || ' empleados ganan menos de 2500');  
FOR i IN ve.FIRST .. ve.LAST  
  LOOP  
    dbms_output.put_line(ve(i).apellidos || ' gana ' || ve(i).salario);  
  END LOOP;
```

END;

/