BASES DE DATOS OBJETO-RELACIONALES Y ORIENTADAS A OBJETOS

1.- Introducción.

Las Bases de Datos Relacionales (BDR) son ideales para aplicaciones tradicionales que soportan tareas administrativas, y que trabajan con datos de estructuras simples y poco cambiantes, incluso cuando la aplicación pueda estar desarrolla en un lenguaje OO y sea necesario un Mapeo Objeto Relacional (ORM).

Pero cuando la aplicación requiere otras necesidades, como por ejemplo, soporte multimedia, almacenar objetos muy cambiantes y complejos en estructura y relaciones, este tipo de base de datos no son las más adecuadas. Recuerda, que si queremos representar un objeto y su relaciones en una BDR esto implica que:

- Los objetos deben ser descompuestos en diferentes tablas.
- A mayor complejidad, mayor número de tablas, de manera que se requieren muchos enlaces (joins) para recuperar un objeto, lo cual disminuye dramáticamente el rendimiento.

Las **Bases de Datos Orientadas a Objetos** (BDOO) o Bases de Objetos se integran directamente y sin problemas con las aplicaciones desarrolladas en lenguajes orientados a objetos, ya que **soportan un modelo de objetos puro** y son ideales para almacenar y recuperar datos complejos permitiendo a los usuarios su **navegación** directa (sin un mapeo entre distintas representaciones).

Las Bases de Objetos aparecieron a finales de los años 80 motivadas fundamentalmente por dos razones:

- Las necesidades de los lenguajes de Bases de Datos Orientadas a Objetos (POO), como la necesidad de persistir objetos.
- Las limitaciones de las bases de datos relacionales, como el hecho de que sólo manejan estructuras muy simples (tablas) y tienen poca riqueza semántica

Pero como las BDOO no terminaban de asentarse, debido fundamentalmente a la inexistencia de un estándar, y las BDR gozaban y gozan en la actualidad de una gran aceptación, experiencia y difusión, debido fundamentalmente a su gran robustez y al lenguaje SQL, los fabricantes de bases de datos comenzaron e implementar nuevas funcionalidades orientadas a objetos en las BDR existentes. Así surgieron, las bases de datos objeto-relacionales.

Las **Bases de Datos Objeto-Relacionales** (BDOR) son bases de datos relacionales que han evolucionado hacia una base de datos más extensa y compleja, incorporando conceptos del modelo orientado a objetos. Pero en estas bases de datos **aún existe un mapeo de objetos subyacente**, que es costoso y poco flexible, cuando los objetos y sus interacciones son complejos.

2.- Características de las bases de datos orientadas a objetos.

En una BDOO, los datos se almacenan como objetos. Un **objeto** es, al igual que en POO, una entidad que se puede identificar unívocamente y que describe tanto el **estado** como el **comportamiento** de una entidad del 'mundo real'. El estado de un objeto se describe mediante atributos y su comportamiento es definido mediante procedimientos o métodos.

Entonces, ¿a qué equivalen las entidades, ocurrencias de entidades y relaciones del modelo relacional? Las entidades son las clases, las ocurrencias de entidad son objetos creados desde las clases, las relaciones se mantienen por medio de inclusión lógica, y no existen claves primarias, los objetos tienen un identificador.

La principal característica de las BDOO es que **soportan un modelo de objetos puro y que el lenguaje de programación y el esquema de la base de datos utilizan las mismas definiciones de tipos**.

Otras características importantes de las BDOO son las siguientes:

- **Soportan las características propias de la Orientación a Objetos** como agregación, encapsulamiento, polimorfismo y herencia. La herencia se mantiene en la propia base de datos.
- **Identificador de objeto** (OID). Cada objeto tiene un identificador, generado por el sistema, que es único para cada objeto, lo que supone que cada vez que se necesite modificar un objeto, habrá que recuperarlo de la base de datos, hacer los cambios y almacenarlo nuevamente. Los OID son independientes del contenido del objeto, esto es, si cambia su información, el objeto sigue teniendo el mismo OID. Dos objetos serán equivalentes si tienen la misma información pero diferentes OID.
- **Jerarquía y extensión de tipos**. Se pueden definir nuevos tipos basándose en otros tipos predefinidos, cargándolos en una jerarquía de tipos (o jerarquía de clases).
- **Objetos complejos**. Los objetos pueden tener una estructura de objeto de complejidad arbitraria, a fin de contener toda la información necesaria que describe el objeto.
- **Acceso navegacional de datos**. Cuando los datos se almacenan en una estructura de red densa y probablemente con una estructura de diferentes niveles de profundidad, el acceso a datos se hace principalmente navegando la estructura de objetos y se expresa de forma natural utilizando las construcciones nativas del lenguaje, sin necesidad de uniones o joins típicas en las BDR.
- **Gestión de versiones**. El mismo objeto puede estar representado por múltiples versiones. Muchas aplicaciones de bases de datos que usan orientación a objetos requieren la existencia de varias versiones del mismo objeto, ya que si estando la aplicación en funcionamiento es necesario modificar alguno de sus módulos, el diseñador deberá crear una nueva versión de cada uno de ellos para efectuar cambios.

2.1.- Ventajas e inconvenientes.

El uso de una BDOO puede ser ventajoso frente a una BD relacional si nuestra aplicación requiere alguno de estos elementos :

- Un gran número de tipos de datos diferentes.
- Un gran número de relaciones entre los objetos.
- Objetos con comportamientos complejos.

Una de las principales ventajas de los sistemas de bases de datos orientados a objetos es la **transparencia**, (manipulación directa de datos utilizando un entorno de programación basado en objetos), por lo que el programador, solo se debe preocupar de los objetos de su aplicación, en lugar de cómo los debe almacenar y recuperar de un medio físico.

Otras ventajas de un sistema de bases de datos orientado a objetos son las siguientes:

- **Gran capacidad de modelado**. El modelado de datos orientado a objetos permite modelar el 'mundo real' de una manera óptima gracias al encapsulamiento y la herencia.
- **Flexibilidad**. Permiten una estructura cambiante con solo añadir subclases.
- **Soporte para el manejo de objetos complejos**. Manipula de forma rápida y ágil objetos complejos, ya que la estructura de la base de datos está dada por referencias (apuntadores lógicos) entre objetos.
- **Alta velocidad de procesamiento**. Como el resultado de las consultas son objetos, no hay que reensamblar los objetos cada vez que se accede a la base de objetos
- **Extensibilidad**. Se pueden construir nuevos tipos de datos a partir de los ya existentes, agrupar propiedades comunes de diversas clases e incluirlas en una superclase, lo que reduce la redundancia.
- **Mejora los costes de desarrollo**, ya que es posible la reutilización de código, una de las características de los lenguajes de programación orientados a objetos.
- **Facilitar el control de acceso y concurrencia**, puesto que se puede bloquear a ciertos objetos, incluso en una jerarquía completa de objetos.
- Funcionan de forma **eficiente en entornos cliente/servidor y arquitecturas distribuidas.**

Pero aunque los sistemas de bases de datos orientados a objetos pueden proporcionar soluciones apropiadas para muchos tipos de aplicaciones avanzadas de bases de datos, también tienen sus **desventajas**. Éstas son las siguientes:

- **Carencia de un modelo de datos universal**. No hay ningún modelo de datos aceptado universalmente, y la mayor parte de los modelos carecen de una base teórica.
- **Falta de estándares**. Existe una carencia de estándares generales para los sistemas de BDOO.
- **Complejidad**. La estructura de una BDOO es más compleja y difícil de entender que la de una BDR.
- **Competencia de otros modelos**. Las bases de datos relacionales y objeto-relacionales están muy asentadas y extendidas, siendo un duro competidor.
- **Difícil optimización de consultas**. La optimización de consultas requiere una compresión de la implementación de los objetos, para poder acceder a la base de

datos de manera eficiente. Sin embargo, esto compromete el concepto de encapsulación.

4.- Características de las bases de datos objeto-relacionales.

Las BDOR las podemos ver como un híbrido de las BDR y las BDOO que intenta aunar los beneficios de ambos modelos, aunque por descontado, ello suponga renunciar a algunas características de ambos.

Los objetivos que persiguen estas bases de datos son:

- Mejorar la representación de los datos mediante la orientación a objetos.
- Simplificar el acceso a datos, manteniendo el sistema relacional.

En una BDOR se siguen almacenando tablas en filas y columnas, aunque la estructura de las filas no está restringida a contener escalares o valores atómicos, sino que las columnas pueden almacenar tipos estructurados (tipos compuestos como vectores, conjuntos, etc.) y las tablas pueden ser definidas en función de otras, que es lo que se denomina herencia directa.

Y eso, ¿cómo es posible?

Pues porque internamente tanto las tablas como las columnas son tratados como objetos, esto es, se realiza un mapeo objeto-relacional de manera transparente.

Como consecuencia de esto, aparecen **nuevas características**, entre las que podemos destacar las siguientes:

- **Tipos definidos por el usuario**. Se pueden crear nuevos tipos de datos definidos por el usuario, y que son compuestos o estructurados, esto es, será posible tener en una columna un atributo multivaluado (un tipo compuesto).
- **Tipos Objeto**. Posibilidad de creación de objetos como nuevo tipo de dato que permiten relaciones anidadas.
- **Reusabilidad**. Posibilidad de guardar esos tipos en el gestor de la BDOR, para reutilizarlos en tantas tablas como sea necesario.
- **Creación de funciones**. Posibilidad de definir funciones y almacenarlas en el gestor. Las funciones pueden modelar el comportamiento de un tipo objeto, en este caso se llaman métodos.
- **Tablas anidadas**. Se pueden definir columnas como arrays o vectores multidimensionales, tanto de tipos básicos como de tipos estructurados, esto es, se pueden anidar tablas
- **Herencia** con subtipos y subtablas.

5. BASES OBJETO-RELACIONALES EN ORACLE

A partir de ahora nos centraremos en el estudio de las características de la orientación a objetos de las bases de datos de Oracle y en el cómo acceder a estos datos.

5.1. TIPOS DE OBJETOS

Para crear tipos de objetos utilizamos la orden **CREATE OR REPLACE TYPE** nombre **AS OBJECT**.

El siguiente ejemplo crea dos tipos de objetos, un tipo dirección con los atributos calle, ciudad y código postal. Y otro tipo PERSONA con los atributos código, nombre, dirección y fecha de nacimiento, y la dirección será del tipo creado anteriormente:

```
CREATE OR REPLACE TYPE DIRECCION AS OBJECT
(
        CALLE VARCHAR2(25),
        CIUDAD VARCHAR2(20),
        CODIGO_POST NUMBER(5)
);
/
CREATE OR REPLACE TYPE PERSONA AS OBJECT
(
        CODIGO NUMBER,
        NOMBRE VARCHAR2(35),
        DIREC DIRECCION,
        FECHA_NAC DATE
);
/
```

Oracle responderá con el mensaje: Tipo creado para cada tipo creado.

Una vez creados podemos usarlos para declarar e inicializar objetos como si se tratase de cualquier otro tipo predefinido, hay que tener en cuenta que al declarar el objeto dentro de un bloque PL/SQL debemos inicializarlo.

El siguiente ejemplo muestra la declaración y uso de los tipos creados anteriormente:

```
DECLARE
        DIR DIRECCION := DIRECCION(NULL,NULL,NULL); --SE INICIALIZA vacío
        P PERSONA := PERSONA(NULL,NULL,NULL,NULL);
        DIR2 DIRECCION; -- SE INICIA CON NEW o NO
        P2 PERSONA; -- SE INICIA CON NEW o NO
BEGIN
        DIR.CALLE:='La Mina, 3';
        DIR.CIUDAD:='Guadalajara';
        DIR.CODIGO_POST:=19001;
        --
        P.CODIGO:=1;
        P.NOMBRE := 'JUAN';
        P.DIREC := DIR;
        P.FECHA_NAC := '10/11/1988';
```

```
        DBMS_OUTPUT.PUT_LINE('NOMBRE: '||P.NOMBRE||' * CALLE:'||
P.DIREC.CALLE);
        --
        --Se puede inicializar aquí dentro del cuerpo
        DIR2 := NEW DIRECCION ('C/Madrid 10','Toledo',45002);
        P2:= NEW PERSONA(2,'JUAN', DIR2, SYSDATE);
        DBMS_OUTPUT.PUT_LINE('NOMBRE: '||P2.NOMBRE||' * CALLE: '||
P2.DIREC.CALLE );

END;
/
```

Estructura DECLARE BEGIN END

No es necesario poner **new**.

Para borrar un tipo usamos la orden **DROP TYPE**.

DROP TYPE nombre_tipo;

Si el tipo se utiliza en alguna declaración, se puede borrar el tipo añadiendo la cláusula FORCE. Sin embargo tendremos errores allí donde se utiliza.(no se recomienda)

DROP TYPE nombre_tipo **FORCE**;

Si un tipo se utiliza en una tabla, y el tipo se ha borrado con FORCE, oracle retira las columnas dependientes del tipo, y esas columnas se vuelven inaccesibles, no aparecen en la tabla. (no se recomienda)

Por ejemplo si quiero borrar el tipo DIRECCION: drop type DIRECCION, no puedo porque otro tipo lo utiliza. Hay que poner drop type DIRECCION force; Sin embargo, si otro tipo utiliza el tipo borrado aparecerá un error, sería el caso del tipo PERSONA

5.2. MÉTODOS

Normalmente cuando creamos un objeto también podemos crear los métodos. Los métodos son procedimientos y funciones que se especifican después de los atributos del objeto. Pueden ser de varios tipos:

- MEMBER: son los métodos que sirven para actuar con los objetos. Pueden ser procedimientos y funciones. Pueden acceder a los atributos del tipo, y para ejecutarlos hay que crear un objeto.
- STATIC: son métodos estáticos independientes de las instancias del objeto. Pueden ser procedimientos y funciones. Estos métodos son operaciones globales que no son de los objetos, sino del tipo. Se ejecutan sin instanciar un objeto, no pueden utilizar los atributos del tipo.
- CONSTRUCTOR: sirve para inicializar el objeto. Se trata de una función cuyos argumentos son los valores de los atributos del objeto y que devuelve el objeto inicializado.

Por cada objeto existe un constructor predefinido por Oracle.

Los parámetros del constructor coinciden con los atributos del tipo de objeto, esto es, los parámetros y los atributos se declaran en el mismo orden y tienen el mismo nombre y tipo.

Los constructores llevarán en la cláusula RETURN la expresión RETURN SELF AS RESULT.

PL/SQL nunca invoca al constructor implícitamente, por lo que el usuario debe invocarlo explícitamente.

El siguiente ejemplo muestra el tipo DIRECCION con la declaración de un procedimiento que asigna valor al atributo CALLE y una función que devuelve el valor del atributo CALLE (antes de ejecutar el siguiente código hemos de borrar los tipos creados anteriormente con la orden DROP TYPE nombretipo):

```
CREATE OR REPLACE TYPE DIRECCION AS OBJECT
(
        CALLE VARCHAR2(25),
        CIUDAD VARCHAR2(20),
        CODIGO_POST NUMBER(5),
        MEMBER PROCEDURE SET_CALLE(C VARCHAR2),
        MEMBER FUNCTION GET_CALLE RETURN VARCHAR2
);
/
```

Si no nos deja modificarlo (porque se usa en otro tipo) lo borramos y lo creamos de nuevo:

Borrado de tipo dirección

```
drop type direccion force;
```

El siguiente ejemplo define un tipo rectángulo con 3 atributos, un constructor que recibe 2 parámetros, un método STATIC y otro MEMBER:

```
CREATE OR REPLACE TYPE RECTANGULO AS OBJECT
(
        BASE NUMBER,
        ALTURA NUMBER,
        AREA NUMBER,
        STATIC PROCEDURE PROC1 (ANCHO INTEGER, ALTO INTEGER),
        MEMBER PROCEDURE PROC2 (ANCHO INTEGER, ALTO INTEGER),
        CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
        RETURN SELF AS RESULT
);
/
```

Una vez creado el tipo con la especificación de los atributos y los métodos crearemos el cuerpo del tipo con la implementación de los métodos, usaremos la instrucción **CREATE OR REPLACE TYPE BODY**:

```
CREATE OR REPLACE TYPE BODY nombre_del_tipo AS
        <Implementación de los métodos>
END;
```

Donde <Implementación de los métodos> tiene el siguiente formato:

```
[STATIC | MEMBER] PROCEDURE nombreProc [(parametro1, parámetro2, ...)]
IS
        Declaraciones;
BEGIN
        Instrucciones;
END;

[STATIC | MEMBER | CONSTRUCTOR] FUNCTION nombreFunc [(param1, param2, ... )]
RETURN tipo_valor_retorno
IS
        Declaraciones;
BEGIN
        Instrucciones;
END;
```

La implementación de los métodos del objeto DIRECCION es la siguiente:

```
CREATE OR REPLACE TYPE BODY DIRECCION AS
--
        MEMBER PROCEDURE SET_CALLE(C VARCHAR2) IS
        BEGIN
                CALLE := C;
        END;
--
        MEMBER FUNCTION GET_CALLE RETURN VARCHAR2 IS
        BEGIN
                RETURN CALLE;
        END;
END;
/
```

El siguiente bloque PL/SQL muestra el uso del objeto DIRECCION, visualizará el nombre de la calle, al no definir constructor es necesario invocarlo al definir el objeto (también se puede llamar al constructor con el operador NEW, no es obligatorio):

```
DECLARE
        DIR DIRECCION := DIRECCION(NULL,NULL,NULL);--Llamada al constructor
BEGIN
        DIR.SET_CALLE('La Mina, 3');
        DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE);
        DIR := NEW DIRECCION ('C/Madrid 10','Toledo',45002);
        DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE);
```

```
END;
/

-- ejemplo de uso
declare
	dir direccion := direccion('C/Bardales 4', 'Talavera', 45600);
begin
	dbms_output.put_line('1-Calle = '||dir.get_CALLE());
	dir.set_calle('Nueva calle');
	dbms_output.put_line('2-Calle = '||dir.get_CALLE());
end;
/
```

La implementación de los métodos del objeto RECTÁNGULO se muestra a continuación; antes se crea la tabla TABLAREC que usarán los métodos para insertar datos.

En el constructor para hacer referencia a los atributos del objeto a partir del cual se invocó el método usamos el cualificador SELF delante del atributo, en el método STATIC no están permitidas las referencias a los atributos de instancia, en los métodos MEMBER sí está permitido:

```
CREATE TABLE TABLAREC (VALOR INTEGER);
/

CREATE OR REPLACE TYPE BODY RECTANGULO AS
--
	CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
RETURN SELF AS RESULT
	IS
	BEGIN
		SELF.BASE := BASE;
		SELF.ALTURA := ALTURA;
		SELF.AREA := BASE * ALTURA;
		RETURN;
	END;

--
	STATIC PROCEDURE PROC1 (ANCHO INTEGER, ALTO INTEGER)
	IS
	BEGIN
		INSERT INTO TABLAREC VALUES(ANCHO*ALTO);
		--ALTURA:=ALTO; --ERROR NO SE PUEDE ACCEDER A LOS ATRIBUTOS
	DEL TIPO
		DBMS_OUTPUT.PUT_LINE('FILA INSERTADA');
		COMMIT;
	END;
--
	MEMBER PROCEDURE PROC2 (ANCHO INTEGER, ALTO INTEGER)
```

```
        IS
        BEGIN
                SELF.ALTURA:=ALTO; --SE PUEDE ACCEDER A LOS ATRIBUTOS DEL
        TIPO
                SELF.BASE:=ANCHO; --se puede omitir self
                AREA:=ALTURA*BASE;
                INSERT INTO TABLAREC VALUES(AREA);
                DBMS_OUTPUT.PUT_LINE('FILA INSERTADA');
                COMMIT;
        END;
END;
/
```

El siguiente bloque PL/SQL muestra el uso del objeto RECTANGULO, se puede llamar al constructor usando los 3 atributos; pero es más robusto llamarlo usando 2 atributos de esta manera nos aseguramos que el atributo AREA tiene el valor inicial correcto.

En este caso no es necesario inicializar los objetos R1 y R2 ya que se inicializan en el bloque BEGIN al llamar al constructor con NEW:

```
DECLARE
        R1 RECTANGULO;
        R2 RECTANGULO;
        R3 RECTANGULO := RECTANGULO(NULL,NULL,NULL);
BEGIN
        R1 := NEW RECTANGULO(10,20,200);
        DBMS_OUTPUT.PUT_LINE('AREA R1:'||R1.AREA);

        R2 := NEW RECTANGULO(10,20);
        DBMS_OUTPUT.PUT_LINE('AREA R2:'||R2.AREA);

        R3.BASE := 5;
        R3.ALTURA := 15;
        R3.AREA := R3.BASE * R3.ALTURA;
        DBMS_OUTPUT.PUT_LINE('AREA R3:'||R3.AREA);

        --USO DE LOS MÉTODOS DEL TIPO RECTANGULO
        RECTANGULO.PROC1(10,20); --LLAMADA AL MÉTODO STATIC
        --RECTANGULO.PROC2(20,30); --ERROR, LLAMADA AL MÉTODO MEMBER
        --R1.PROC1(5, 6); --ERROR, LLAMADA AL MÉTODO STATIC
        R1.PROC2(5, 10); --LLAMADA AL MÉTODO MEMBER
END;
/
```

En cuanto a los métodos, se produce error al llamar al método STATIC usando cualquiera de los objetos instanciados.

```
RECTANGULO.PROC1(10,20); -- CORRECTO. LLAMADA SIN INSTANCIAR
```

R1.PROC1(5, 6); --ERROR. Llamada con una instancia

También se produce error si la llamada a un método MEMBER se realiza sin haber instanciado un objeto.

RECTANGULO.PROC2(20,30); --ERROR, LLAMADA SIN INSTANCIAR
R1.PROC2(5, 10); --CORRECTO. Llamada con una instancia

Para borrar el cuerpo de un tipo usamos la orden **DROP TYPE BODY** indicando a la derecha el nombre del tipo cuyo cuerpo deseamos borrar:

DROP TYPE BODY nombre_tipo.

EJEMPLO DEL TIPO RECTÁNGULO CON DOS CONSTRUCTORES, CON DIFERENTES PARÁMETROS

```
create or replace TYPE RECTANGULO AS OBJECT
(
	BASE NUMBER,
	ALTURA NUMBER,
	AREA NUMBER,
	CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
RETURN SELF AS RESULT,
	CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER) RETURN SELF AS
RESULT
);
/

create or replace TYPE BODY RECTANGULO AS CONSTRUCTOR FUNCTION
RECTANGULO (BASE NUMBER, ALTURA NUMBER) --CONSTRUCTOR 1
	RETURN SELF AS RESULT
	AS
	BEGIN
		SELF.BASE := BASE;
		SELF.ALTURA := ALTURA;
		SELF.AREA := BASE * ALTURA;
		RETURN;
	END;

CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER) --CONSTRUCTOR 2
	RETURN SELF AS RESULT
	AS
		BEGIN
		SELF.BASE := BASE;
		SELF.ALTURA := BASE;
		SELF.AREA := BASE * BASE;
		RETURN;
	END;
```

```
END;
/

--Ejemplo de uso

DECLARE
        R1 RECTANGULO;
        R2 RECTANGULO;
        R3 RECTANGULO := RECTANGULO(NULL,NULL,NULL);
        R4 RECTANGULO;
BEGIN
        R1 := NEW RECTANGULO(10,20,200);
        DBMS_OUTPUT.PUT_LINE('AREA R1:'||R1.AREA);
        R2 := NEW RECTANGULO(10,20);
        DBMS_OUTPUT.PUT_LINE('AREA R2:'||R2.AREA);
        R3.BASE := 5;
        R3.ALTURA := 15;
        R3.AREA := R3.BASE * R3.ALTURA;
        DBMS_OUTPUT.PUT_LINE('AREA R3:'||R3.AREA);
        R4 := NEW RECTANGULO(10);
        DBMS_OUTPUT.PUT_LINE('AREA R4:'||R4.AREA);
END;
/
```

5.3. COMPARACIÓN DE TIPOS

En muchas ocasiones necesitamos comparar en incluso ordenar datos de tipos definidos como OBJECT. Para ello es necesario crear un método **MAP** u **ORDER**, debiéndose definir al menos uno de ellos por cada objeto que se quiere comparar:

- Los métodos MAP consisten en una función que devuelve un valor de tipo escalar (CHAR, VARCHAR2, NUMBER, DATE, …) que será el que se utilice en las comparaciones y ordenaciones aplicando los criterios establecidos para este tipo de datos.
- Un método ORDER utiliza los atributos del objeto sobre el que se ejecuta para realizar un cálculo y compararlo con otro objeto del mismo tipo que toma como argumento de entrada. Este método devuelve un valor negativo si el parámetro de entrada es mayor que el atributo, un valor positivo si ocurre lo contrario y cero si ambos son iguales. Suelen ser menos funcionales y eficientes, se utilizan cuando el criterio de comparación es muy complejo como para implementarlo con un método MAP. No lo trataremos en este tema.

Por ejemplo la siguiente declaración indica que los objetos de tipo PERSONA se van a comparar por su atributo CODIGO:

```
CREATE OR REPLACE TYPE PERSONA AS OBJECT
(
        CODIGO NUMBER,
        NOMBRE VARCHAR2(35),
```

```
        DIREC DIRECCION,
        FECHA_NAC DATE,
        MAP MEMBER FUNCTION POR_CODIGO RETURN NUMBER
);
/
CREATE OR REPLACE TYPE BODY PERSONA AS
        MAP MEMBER FUNCTION POR_CODIGO RETURN NUMBER IS
        BEGIN
                RETURN CODIGO;
        END;
END;
/

--Utilizando un método order podríamos comparar así:

create or replace TYPE PERSONA2 AS OBJECT
(
        CODIGO NUMBER,
        NOMBRE VARCHAR2(35),
        DIREC DIRECCION,
        FECHA_NAC DATE,
        order MEMBER FUNCTION comparar ( pe PERSONA2) RETURN NUMBER
);
/

create or replace TYPE BODY PERSONA2 AS
        order MEMBER FUNCTION comparar ( pe PERSONA2) RETURN NUMBER
        is
        BEGIN
                If codigo < pe.codigo then return -1; end if;
                If codigo > pe.codigo then return 1; end if;
                If codigo = pe.codigo then return 0; end if;
        END;
end;
/
```

El siguiente código PL/SQL compara dos objetos de tipo PERSONA, y visualiza ‘OBJETOS IGUALES’ ya que el atributo CODIGO tiene el mismo valor para los dos objetos:

```
DECLARE
        P1 PERSONA := PERSONA(NULL,NULL,NULL,NULL);
        P2 PERSONA:= PERSONA(NULL,NULL,NULL,NULL);
BEGIN
        P1.CODIGO := 1;
        P1.NOMBRE := 'JUAN';
        P2.CODIGO :=1;
        P2.NOMBRE :='MANUEL';
        IF P1= P2 THEN
```

```
            DBMS_OUTPUT.PUT_LINE('OBJETOS IGUALES');
        ELSE
            DBMS_OUTPUT.PUT_LINE('OBJETOS DISTINTOS');
        END IF;
END;
/
```

En este caso se ejecuta el método order, no hay ninguna diferencia, el segundo objeto lo toma como argumento de entrada.

```
DECLARE
        P1 PERSONA2 := PERSONA2(NULL,NULL,NULL,NULL);
        P2 PERSONA2 := PERSONA2(NULL,NULL,NULL,NULL);
BEGIN
        P1.CODIGO := 1;
        P1.NOMBRE := 'JUAN';
        P2.CODIGO :=1;
        P2.NOMBRE :='MANUEL';
        IF P1 = P2 THEN --P2 hace de argumento de entrada
            DBMS_OUTPUT.PUT_LINE('OBJETOS IGUALES');
        ELSE
            DBMS_OUTPUT.PUT_LINE('OBJETOS DISTINTOS');
        END IF;
END;
/
```

Es necesario un método MAP u ORDER para comparar objetos en PL/SQL.

Un tipo de objeto sólo puede tener un método MAP o uno ORDER.

4.3. TABLAS DE OBJETOS

Una tabla de objetos es una tabla que almacena un objeto en cada fila, se accede a los atributos de esos objetos como si se tratasen de columnas de la tabla.

El siguiente ejemplo crea la tabla ALUMNOS de tipo PERSONA con la columna CODIGO como clave primaria y muestra su descripción:

```
CREATE TABLE ALUMNOS OF PERSONA (
        CODIGO PRIMARY KEY
);
/
DESC ALUMNOS;
Nombre                                    Nulo         Tipo
 ----------------------------------------- -------------- ------------------
CODIGO                                    NOT NULL NUMBER
NOMBRE                                                     VARCHAR2(35)
```

DIREC DIRECCION
FECHA_NAC DATE

A continuación se insertan filas en la tabla ALUMNOS. Hemos de poner delante el tipo (DIRECCION) a la hora de dar valores a los atributos que forman la columna de dirección:

```
INSERT INTO ALUMNOS VALUES(
        1, 'Juan Pérez', DIRECCION ('C/Los manantiales 5', 'GUADALAJARA', 19005),
        '18/12/1991'
);

INSERT INTO ALUMNOS VALUES(
        Persona(11, 'Juan Pérez', DIRECCION ('C/Los manantiales 5', 'GUADALAJARA',
        19005), '18/12/1991')
);

INSERT INTO ALUMNOS (CODIGO, NOMBRE, DIREC, FECHA_NAC) VALUES (
        2, 'Julia Breña', DIRECCION ('C/Los espartales 25', 'GUADALAJARA', 19004),
        '18/12/1987'
);
```

El siguiente bloque PL/SQL inserta una fila en la tabla ALUMNOS:

```
DECLARE
        DIR DIRECCION:= DIRECCION ('C/Sevilla 20', 'GUADALAJARA', 19004);
        PER PERSONA:= PERSONA(5,'MANUEL',DIR, '20/10/1987');
BEGIN
        INSERT INTO ALUMNOS VALUES(PER); --insertar
        COMMIT;
END;
/
```

ACCESO A UNA TABLA DE OBJETOS DESDE JAVA

Los objetos se devuelven en tipos de datos **Struct**, y los atributos de esta Struct se devuelven en un array de objetos:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Struct;

public class Acceso {
        public static void main(String[] args) {
                try {
// Oracle Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
                        Connection conexion =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "system",
                                        "123456789");

// Preparamos la consulta
                        Statement sentencia = conexion.createStatement();
                        String sql = "select * from ALUMNOS";
                        ResultSet resul = sentencia.executeQuery(sql);

                        while (resul.next()) {
                                int CODIGO = resul.getInt(1);
                                String NOMBRE = resul.getString(2);
                                java.sql.Date FECHA_NAC = resul.getDate(4);

// Obtengo el objeto DIRECCION
                                Struct objeto = (Struct) resul.getObject(3);

// Saco sus atributos CALLE, CIUDAD, CODIGO_POST
                                Object[] atributos = objeto.getAttributes();
                                String calle = (String) atributos[0];
                                String ciudad = (String) atributos[1];
                                java.math.BigDecimal codigo_post = (java.math.BigDecimal)
atributos[2];

                                System.out.printf("Codigo:%d, Nombre:%s Fecha_nac: %s %n
\t Calle: %s, Ciudad: %s, CP: %d %n", CODIGO,
                                                NOMBRE, FECHA_NAC, calle, ciudad,
codigo_post.intValue());
                        }

                        resul.close(); // Cerrar ResultSet
                        sentencia.close(); // Cerrar Statement
                        conexion.close(); // Cerrar conexión
                } catch (SQLException e) {
                        e.printStackTrace();
                }
        }
}
```

CONSULTAS SOBRE TABLAS DE OBJETOS

Veamos algunos ejemplos de consultas sobre la tabla:

Seleccionar aquellas filas cuya CIUDAD = ‘GUADALAJARA’:

```
SELECT * FROM ALUMNOS A WHERE A.DIREC.CIUDAD = 'GUADALAJARA';
```

Para seleccionar columnas individuales, si la columna es un tipo OBJECT se necesita definir un alias para la tabla; en una base de datos con tipos y objetos se recomienda usar alias para el nombre de las tablas. A continuación seleccionamos el codigo y la dirección de los alumnos:

```
SELECT CODIGO, A.DIREC FROM ALUMNOS A;
```

Para llamar a los métodos hay que utilizar su nombre y paréntesis que encierren los argumentos de entrada (aunque no tenga argumentos los paréntesis deben aparecer).

En el siguiente ejemplo obtenemos el nombre y la calle de los alumnos, usamos el método GET_CALLE del tipo DIRECCION:

```
SELECT NOMBRE, A.DIREC.GET_CALLE() FROM ALUMNOS A;
```

Modificamos aquellas filas cuya ciudad es GUADALAJARA, convertimos la ciudad a minúscula:

```
UPDATE ALUMNOS A SET A.DIREC.CIUDAD=LOWER(A.DIREC.CIUDAD) WHERE
A.DIREC.CIUDAD ='GUADALAJARA';
```

Eliminamos aquellas filas cuya ciudad sea ‘guadalajara’:

```
DELETE ALUMNOS A WHERE A.DIREC.CIUDAD='guadalajara';
```

El siguiente bloque PL/SQL muestra el nombre y la calle de los alumnos:

```
DECLARE
        CURSOR C1 IS SELECT * FROM ALUMNOS;
BEGIN
        FOR I IN C1 LOOP
                DBMS_OUTPUT.PUT_LINE(I.NOMBRE || ' - Calle: '|| I.DIREC.CALLE);
        END LOOP;
END;
/
```

El siguiente bloque PL/SQL modifica la dirección completa de un alumno:

```
DECLARE
        D DIRECCION:= DIRECCION ('C/Galiano 5','Guadalajara',19004);
BEGIN
        UPDATE ALUMNOS
                SET DIREC = D WHERE NOMBRE ='Juan Pérez';
        COMMIT;
END;
/
```

También podría hacer lo siguiente:

```
UPDATE ALUMNOS
        SET DIREC = DIRECCION ('C/Galiano 5','Guadalajara',19004)
        WHERE NOMBRE ='Juan Pérez';
```

Si sólo quiero actualizar la calle:

```
UPDATE ALUMNOS a
        SET a.DIREC.calle='NUEVA CALLE'
        WHERE a.NOMBRE ='Juan Pérez';
```

4.4. TIPOS COLECCIÓN

Las bases de datos relacionales orientadas a objetos pueden permitir el almacenamiento de colecciones de elementos en una única columna. Tal es el caso de los **VARRAYS** en Oracle que son similares a los arrays de Java que permiten almacenar un conjunto de elementos, todos del mismo tipo, y cada elemento tiene un índice asociado; y de las **tablas anidadas** que permiten almacenar en una columna de una tabla otra tabla.

4.4.1. VARRAYS

Para crear una colección de elementos varrays se usa la orden **CREATE TYPE**. El siguiente ejemplo crea un tipo VARRAY de nombre TELEFONO de tres elementos donde cada elemento es del tipo VARCHAR2:

```
CREATE TYPE TELEFONO AS VARRAY(3) OF VARCHAR2(9);
```

Cuando se declara un tipo VARRAY no se produce ninguna reserva de espacio. Para obtener información de un VARRAY usamos la orden **DESC** (DESC TELEFONO). La vista **USER_VARRAYS** obtiene información de las tablas que tienen columnas varrays.

Veamos algunos ejemplos del uso de varrays. Creamos una tabla donde una columna es de tipo VARRAY:

```
CREATE TABLE AGENDA
(
        NOMBRE VARCHAR2(15),
        TELEF TELEFONO
);
```

Insertamos varias filas:

```
INSERT INTO AGENDA VALUES
('MANUEL', TELEFONO ('656008876', '927986655', '639883300'));

INSERT INTO AGENDA (NOMBRE, TELEF) VALUES
('MARTA', TELEFONO ('649500800'));
```

En las consultas es imposible poner condiciones sobre los elementos almacenados dentro del VARRAY, además los valores del VARRAY sólo pueden ser accedidos y recuperados como bloque, no se puede acceder individualmente a los elementos (desde un programa PL/SQL sí se puede). La primera posición es la 1.

Seleccionamos determinadas columnas:

siempre selecciona todo el bloque y devuelve un bloque

```
SELECT TELEF FROM AGENDA;
```

Podemos usar alias para seleccionar las columnas:

```
SELECT A.TELEF FROM AGENDA A;
```

se sobrescribe todo el var array

Modificamos los teléfonos de MARTA:

```
UPDATE AGENDA SET TELEF=TELEFONO('649500800', '659222222')
WHERE NOMBRE = 'MARTA';
```

Desde un programa PL/SQL se puede hacer un bucle para recorrer los elementos del VARRAY.

El siguiente bloque visualiza los nombres y los teléfonos de la tabla AGENDA, I.TELEF.COUNT devuelve el número de elementos del VARRAY:

```
DECLARE
        CURSOR C1 IS SELECT * FROM AGENDA;
        CAD VARCHAR2(50);
BEGIN
        FOR I IN C1 LOOP
                DBMS_OUTPUT.PUT_LINE(I.NOMBRE || ', Número de Telefonos:
                '||I.TELEF.COUNT);
                CAD:='*';
                --Recorro el varray
                FOR J IN 1 .. I.TELEF.COUNT LOOP
                        CAD := CAD ||I.TELEF(J)||'*';
                END LOOP;
                DBMS_OUTPUT.PUT_LINE(CAD);
        END LOOP;
END;
/
```

Muestra la siguiente salida:

```
MANUEL, Número de Telefonos: 3
*656008876*927986655*639883300*
MARTA, Número de Telefonos:2
*649500800*659222222*
```

El siguiente ejemplo crea un procedimiento almacenado para insertar datos en la tabla AGENDA, a continuación, se muestra la llamada al procedimiento:

```
CREATE OR REPLACE PROCEDURE INSERTAR_AGENDA (N VARCHAR2, T
TELEFONO) AS
BEGIN
        INSERT INTO AGENDA VALUES (N,T);
END;
/
BEGIN
        INSERTAR_AGENDA('LUIS', TELEFONO('949009977'));
        INSERTAR_AGENDA('MIGUEL', TELEFONO('949004020', '678905400'));
        COMMIT;
END;
/
```

Para obtener información sobre la colección tenemos los siguientes métodos:

Parámetros	**Función**
COUNT	Devuelve el número de elementos de una colección.
EXISTS	Devuelve TRUE si la fila existe.
FIRST/LAST	Devuelve el índice del primer y último elemento de la colección.
NEXT/PRIOR	Devuelve el elemento siguiente o anterior del actual.
LIMIT	Informa del número máximo de elementos que puede contener la colección.

Para modificar los elementos de la colección tenemos los siguientes métodos:

Parámetros	Función
DELETE	Elimina todos los elementos de una colección.
EXTEND	Añade un elemento nulo a la colección.
EXTEND(n)	Añade n elementos nulos.
TRIM	Elimina el elemento situado al final de la colección.

TRIM(n)	Elimina n elementos del final de la colección.

El siguiente ejemplo muestra cómo usar los parámetros:

```
DECLARE
	TEL TELEFONO := TELEFONO(NULL, NULL, NULL);
BEGIN
	SELECT TELEF INTO TEL FROM AGENDA WHERE NOMBRE = 'MARTA';
	--Visualizar Datos
	DBMS_OUTPUT.PUT_LINE('Nº DE TELÉFONOS ACTUALES: ' || TEL.COUNT);
	DBMS_OUTPUT.PUT_LINE('ÍNDICE DEL PRIMER ELEMENTO: ' || TEL.FIRST);
	DBMS_OUTPUT.PUT_LINE('ÍNDICE DEL ÚLTIMO ELEMENTO: ' || TEL.LAST);
	DBMS_OUTPUT.PUT_LINE('MÁXIMO Nº DE TLFS PERMITIDO:' || TEL.LIMIT);

	--Añade un número de teléfono a MARTA
	TEL.EXTEND;
	TEL(TEL.COUNT):= '123000000';
	UPDATE AGENDA A SET A.TELEF = TEL WHERE NOMBRE = 'MARTA';

	--Elimina un teléfono
	SELECT TELEF INTO TEL FROM AGENDA WHERE NOMBRE = 'MANUEL';
	TEL.TRIM; --Elimina el último elemento del array
	TEL.DELETE; --Elimina todos los elementos
	UPDATE AGENDA A SET A.TELEF = TEL WHERE NOMBRE = 'MANUEL';
END;
/
```

Código Java para acceder a la tabla AGENDA

Los colecciones de elementos se devuelven en tipos de datos **Array**, del paquete java.sql, y los elementos de la colección que forman parte del Array se devuelven en un array de objetos:

```
import java.sql.Array;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Acceso {
	public static void main(String[] args) {
		try {
			// Oracle
			//Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
                        Connection conexion =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "system",
"123456789");

                        // Preparamos la consulta
                        Statement sentencia = conexion.createStatement();
                        String sql = "select * from AGENDA";
                        ResultSet resul = sentencia.executeQuery(sql);

                        while (resul.next()) {
                                String NOMBRE = resul.getString(1);
                                System.out.print("Nombre: " + NOMBRE + " => ");
                                // Obtengo el array de los telefonos
                                try {
                                        Array objeto = (Array) resul.getObject(2);
                                        Object[] telefonos = (Object[]) objeto.getArray();

                                        if (telefonos.length == 0)
                                                System.out.printf("NO TIENE NINGÚN
TELÉFONO EN EL ARRAY \n");
                                        else {
                                                for (int i = 0; i < telefonos.length; i++) {
                                                        try {
                                                                System.out.printf("%s ",
telefonos[i].toString());
                                                        } catch (java.lang.NullPointerException
n) {
                                                                System.out.print("nulo ");
                                                        }
                                                }
                                                System.out.println();
                                        }
                                } catch (java.lang.NullPointerException n) {
                                        System.out.printf("NO TIENE TELEFONOS - ARRAY
NULL \n");
                                }
                        }

                        resul.close(); // Cerrar ResultSet
                        sentencia.close(); // Cerrar Statement
                        conexion.close(); // Cerrar conexión
                } catch (SQLException e) {
                        e.printStackTrace();
                }
        }
}
```

tabla anidada

4.4.2. TABLAS ANIDADAS

Una tabla anidada está formada por un conjunto de elementos, todos del mismo tipo.

La tabla anidada está contenida en una columna y el tipo de esta columna debe ser un tipo de objeto existente en la base de datos.

Para crear una tabla anidada usamos la orden **CREATE TYPE**. Sintaxis:

CREATE TYPE nombre_tipo AS TABLE OF tipo_de_dato;

El siguiente ejemplo crea un tipo tabla anidada que almacenará objetos del tipo DIRECCION (creado al principio de la unidad):

CREATE TYPE TABLA_ANIDADA AS TABLE OF DIRECCION;

No es necesario especificar el tamaño máximo de una tabla anidada. Veamos como se define una columna de una tabla con el tipo tabla anidada creada anteriormente:

```
CREATE TABLE EJEMPLO_TABLA_ANIDADA
(
        ID NUMBER(2),
        APELLIDOS VARCHAR2(35),
        DIREC TABLA_ANIDADA
)
NESTED TABLE DIREC STORE AS DIREC_ANIDADA;
```

La cláusula **NESTED TABLE** identifica el nombre de la columna que contendrá la tabla anidada.

La cláusula **STORE AS** especifica el nombre de la tabla (DIREC_ANIDADA) en la que se van a almacenar las direcciones que se representan en el atributo DIREC de cualquier objeto de la tabla EJEMPLO_TABLA_ANIDADA.
La descripción del tipo TABLA_ANIDADA y de la tabla EJEMPLO_TABLA_ANIDADA es la siguiente:

```
SQL> DESC TABLA_ANIDADA;
TABLA_ANIDADA TABLE OF DIRECCION
Nombre                                    Nulo        Tipo
 ----------------------------------------- ---------- ----------------------
CALLE                                                 VARCHAR2(25)
CIUDAD                                                VARCHAR2(20)
CODIGO_POST                                           NUMBER(5)

METHOD
------
MEMBER PROCEDURE SET_CALLE
Nombre de Argumento             Tipo                    E/S  Por Defecto
```

```
---------------------------------------- ---------------------- ------ ---------------
C                                        VARCHAR2         IN

METHOD
------
MEMBER FUNCTION GET_CALLE RETURNS VARCHAR2

SQL> DESC EJEMPLO_TABLA_ANIDADA ;

Nombre                                   Nulo  Tipo
---------------------------------------- ------ --------------------
ID                                              NUMBER(2)
APELLIDOS                                       VARCHAR2(35)
DIREC                                           TABLA_ANIDADA
```

Veamos algunos ejemplos con la tabla.

Insertamos varias filas con varias direcciones en la tabla EJEMPLO_TABLA_ANIDADA:

```
INSERT INTO EJEMPLO_TABLA_ANIDADA VALUES (1, 'RAMOS',
        TABLA_ANIDADA (
                DIRECCION ('C/Los manantiales 5', 'GUADALAJARA', 19004),
                DIRECCION ('C/Los manantiales 10', 'GUADALAJARA', 19004),
                DIRECCION ('C/Av de Paris 25', 'CÁCERES', 10005),
                DIRECCION ('C/Segovia 23-3A', 'TOLEDO', 45005)
        )
);

INSERT INTO EJEMPLO_TABLA_ANIDADA VALUES (2, 'MARTÍN',
        TABLA_ANIDADA (
                DIRECCION ('C/Huesca 5', 'ALCALÁ DE H', 28804),
                DIRECCION ('C/Madrid 20', 'ALCORCÓN', 28921)
        )
);
```

Se inserta el código, el nombre y la tabla anidada vacía:

```
INSERT INTO EJEMPLO_TABLA_ANIDADA
        VALUES (5, 'PEREZ', TABLA_ANIDADA());
```

Seleccionamos todas las filas de la tabla:

```
SELECT * FROM EJEMPLO_TABLA_ANIDADA;
```

El siguiente ejemplo obtiene el identificador, el apellido y la dirección completa, de todas las filas de la tabla. Se obtienen tantas filas como calles tiene cada identificador. El **operador TABLE** con la columna que es tabla anidada entre paréntesis y colocado a la derecha de

FROM se utiliza para acceder a todas las filas de la tabla anidada, es necesario indicar el alias (en este caso se llama DIRECCION); con DIRECCION.* se obtienen todos los campos de la dirección:

```
SELECT ID, APELLIDOS, DIRECCION.*
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) DIRECCION;
```

En la consulta anterior la columna que es tabla anidada se utiliza como si fuese una tabla normal, incluyéndola en la cláusula FROM.

El siguiente ejemplo obtiene las direcciones completas del identificador 1:

```
SELECT ID, DIRECCION.*
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) DIRECCION WHERE ID=1;
```

INSERTAR DATOS EN TABLA ANIDADA

Insertamos una dirección al final de la tabla anidada para el identificador 1 (ahora el identificador 1 tendrá cinco direcciones):

```
INSERT INTO TABLE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1)
VALUES (DIRECCION ('C/Los manantiales 15', 'GUADALAJARA', 19004));
```

La cláusula TABLE a la derecha de INTO se utiliza para acceder a la fila que nos interesa, en este caso la que tiene ID=1.

MODIFICAR EN TABLA ANIDADA

En el siguiente ejemplo se modifica la primera dirección del identificador 1, se le asigna el valor 'C/Pilon 11', 'TOLEDO', 45589:

```
UPDATE TABLE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1)  PRIMERA
SET VALUE(PRIMERA) = DIRECCION ('C/Pilon 11', 'TOLEDO', 45589)
WHERE
VALUE(PRIMERA)= DIRECCION('C/Los manantiales 5', 'GUADALAJARA', 19004);
```

El alias PRIMERA recoge los datos devueltos por la SELECT (que debe devolver una fila). Con SET VALUE (PRIMERA) se asigna el valor 'C/Pilon 11', 'TOLEDO', 45589 al objeto DIRECCIÓN cuyo valor coincida con 'C/Los manantiales 5', 'GUADALAJARA', 19004; esto se indica en la cláusula WHERE con la función VALUE(PRIMERA).

En el siguiente ejemplo se modifican (para el identificador 1) todas las direcciones que tengan la ciudad de GUADALAJARA, se le asigna el valor MADRID. En este caso no se necesita a función VALUE ya que se modifica la columna CIUDAD y no un objeto:

```
UPDATE TABLE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
SET PRIMERA.CIUDAD = 'MADRID'
WHERE PRIMERA.CIUDAD = 'GUADALAJARA';
```

BORRAR EN TABLA ANIDADA

En el siguiente ejemplo se elimina la segunda dirección del identificador 1, aquella cuyo valor es 'C/Los manantiales 10', 'GUADALAJARA', 19004:

```
DELETE FROM TABLE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE
VALUE(PRIMERA)=DIRECCION('C/Los manantiales 10', 'GUADALAJARA', 19004);
```

En el siguiente ejemplo se eliminan todas las direcciones del identificador 1 con ciudad igual a 'GUADALAJARA':

```
DELETE FROM TABLE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE
PRIMERA.CIUDAD ='GUADALAJARA';
```

CÓDIGO JAVA PARA ACCEDER A LA TABLA ANIDADA:

Las tablas anidada se devuelven en tipos de datos **Array**, del paquete java.sql, y los elementos de la colección que forman parte del Array se devuelven en un array de objetos. Para obtener los objetos que formen parte de la tabla anidada habrá que utilizar tipos de datos **Struct**:

```
        import java.sql.Array;
        import java.sql.Connection;
        import java.sql.DriverManager;
        import java.sql.ResultSet;
        import java.sql.SQLException;
        import java.sql.Statement;
        import java.sql.Struct;


        public class Acceso {

                public static void main(String[] args) {
                        try {
                                // Oracle
                                Connection conexion =
        DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "system",
                                                "123456789");
```

```
                        // Preparamos la consulta
                        Statement sentencia = conexion.createStatement();
                        String sql = "select * from EJEMPLO_TABLA_ANIDADA ";
                        ResultSet resul = sentencia.executeQuery(sql);

                        while (resul.next()) {
                                int ID = resul.getInt(1);
                                String APELLIDOS = resul.getString(2);
                                System.out.println("ID: " + ID + ", Apellidos: " +
APELLIDOS + " => ");

                                // extraer columna DIREC TABLA_ANIDADA()
                                try {
                                        Array DIREC = (Array) resul.getObject(3);
                                        Object[] direcciones = (Object[])
DIREC.getArray();
                                        if (direcciones.length == 0)
                                                System.out.printf("\tNO TIENE
NINGUNA DIRECCIÓN - TABLA ANIDADA VACIA \n");
                                        else {
                                                for (int i = 0; i < direcciones.length; i++) {
                                                        try {
                                                                Struct unadireccion =
(Struct) direcciones[i];
                                                                // Saco sus atributos
CALLE, CIUDAD, //CODIGO_POST
                                                                Object[] atributos =
unadireccion.getAttributes();
                                                                String calle = (String)
atributos[0];
                                                                String ciudad = (String)
atributos[1];
                                                                java.math.BigDecimal
codigo_post = (java.math.BigDecimal) atributos[2];
                                                                System.out.printf("\t
Calle: %s, Ciudad: %s, CP: %d %n", calle, ciudad,
codigo_post.intValue());
                                                        } catch
(java.lang.NullPointerException n) {
                                                                System.out.print("nula ");
                                                        }
                                                }
                                                System.out.println();
                                        }
                                } catch (java.lang.NullPointerException n) {
```

```
                                        System.out.printf("\tNO TIENE DIRECCIONES -
    TABLA ANIDADA NULL \n");
                                    }
                                }

                                resul.close(); // Cerrar ResultSet
                                sentencia.close(); // Cerrar Statement
                                conexion.close(); // Cerrar conexión
                            } catch (SQLException e) {
                                e.printStackTrace();
                            }
                        }
                }
```

4.2.5. REFERENCIAS

Mediante el **operador REF** asociado a un atributo se pueden definir referencias a otros objetos. Un atributo de este tipo almacena una referencia al objeto del tipo definido e implementa una relación de asociación entre los dos tipos de objetos.

Una columna de tipo REF guarda un puntero a una fila de la otra tabla, contiene el OID (identificador del objeto fila) de dicha fila.

El siguiente ejemplo crea un tipo EMPLEADO_T donde uno de los atributos es una referencia a un objeto EMPLEADO_T, después se crea una tabla de objetos EMPLEADO_T:

```
CREATE TYPE EMPLEADO_T AS OBJECT (
        NOMBRE VARCHAR2(30),
        JEFE REF EMPLEADO_T
);
/
CREATE TABLE EMPLEADO OF EMPLEADO_T;
```

Insertamos filas en la tabla, el segundo INSERT asigna al atributo JEFE la referencia al objeto con apellido GIL:

```
INSERT INTO EMPLEADO VALUES (EMPLEADO_T ('GIL', NULL));

INSERT INTO EMPLEADOSELECT EMPLEADO_T ('ARROYO', REF(E)) FROM
EMPLEADO E WHERE E.NOMBRE = 'GIL';

INSERT INTO EMPLEADOSELECT EMPLEADO_T ('RAMOS', REF(E)) FROM EMPLEADO
E WHERE E.NOMBRE = 'GIL';
```

Para acceder al objeto referido por un REF se utiliza el **operador DEREF**, en el ejemplo se visualiza el nombre del empleado y los datos del jefe de cada empleado:

```
SELECT NOMBRE, DEREF(P.JEFE) FROM EMPLEADO P;

NOMBRE          DEREF(P.JEFE)(NOMBRE, JEFE)
—---------------- ----------------------------------------------------
GIL
ARROYO          EMPLEADO_T('GIL', NULL)
RAMOS           EMPLEADO_T('GIL', NULL)
```

También se puede omitir DEREF

```
SELECT NOMBRE, DEREF(P.JEFE) FROM EMPLEADO P;
SELECT NOMBRE, P.JEFE FROM EMPLEADO P;
```

La siguiente consulta obtiene el identificador del objeto cuyo nombre es GIL:

```
SELECT REF(P) FROM EMPLEADO P WHERE NOMBRE='GIL';
```

La siguiente consulta obtiene nombre del empleado y el nombre de su jefe:

```
SQL> SELECT NOMBRE, DEREF(P.JEFE).NOMBRE FROM EMPLEADO P;

NOMBRE                    DEREF(P.JEFE).NOMBRE
------------------------- ------------------------------------
GIL
ARROYO                     GIL
RAMOS                     GIL
```

También se puede omitir DEREF

```
SELECT NOMBRE, DEREF(P.JEFE).nombreFROM EMPLEADO P;
SELECT NOMBRE, P.JEFE.nombreFROM EMPLEADO P;
```

El siguiente ejemplo actualiza el jefe del nombre RAMOS, se le asigna ARROYO:

```
UPDATE EMPLEADO SET JEFE =
(SELECT REF(E) FROM EMPLEADO E WHERE NOMBRE='ARROYO')
WHERE NOMBRE='RAMOS'.
```

4.2.6. HERENCIA DE TIPOS

La herencia facilita la creación de objetos a partir de otros ya existentes e implica que un subtipo obtenga todo el comportamiento (métodos) y eventualmente los atributos de su supertipo. Los subtipos definen sus propios atributos y métodos y puede redefinir los métodos que heredan, esto se conoce como polimorfismo. El siguiente ejemplo define un tipo persona y a continuación el subtipo tipo alumno:

```
-- Se define el tipo persona
--
CREATE OR REPLACE TYPE TIPO_PERSONA AS OBJECT(
        DNI VARCHAR2(10),
        NOMBRE VARCHAR2(25),
        FEC_NAC DATE,
        MEMBER FUNCTION EDAD RETURN NUMBER,
        FINAL MEMBER FUNCTION GET_DNI
                                        RETURN VARCHAR2, -- No se puede redefinir
        MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2,
        MEMBER PROCEDURE VER_DATOS
) NOT FINAL;-- Se pueden derivar subtipos
/
-- Cuerpo del tipo persona
--
CREATE OR REPLACE TYPE BODY TIPO_PERSONA AS
MEMBER FUNCTION EDAD RETURN NUMBER IS
        ED NUMBER;
BEGIN
        ED := TO_CHAR(SYSDATE, 'YYYY') - TO_CHAR(FEC_NAC, 'YYYY');
        RETURN ED;
END;
--
FINAL MEMBER FUNCTION GET_DNI RETURN VARCHAR2 IS
BEGIN
        RETURN DNI;
END;
--
MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2 IS
BEGIN
        RETURN NOMBRE;
END;
--
MEMBER PROCEDURE VER_DATOS IS
BEGIN
        DBMS_OUTPUT.PUT_LINE(DNI|| '*'||NOMBRE||'*'||EDAD());
END;
END;
/
-- Se define el tipo alumno
--
CREATE OR REPLACE TYPE TIPO_ALUMNO UNDER TIPO_PERSONA(
-- se define un subtipo
        CURSO VARCHAR2(10),
        NOTA_FINAL NUMBER,
        MEMBER FUNCTION NOTA RETURN NUMBER,
        OVERRIDING MEMBER PROCEDURE VER_DATOS -- se redefine ese método
);
```

```
/ -- Cuerpo del tipo alumno
--
CREATE OR REPLACE TYPE BODY TIPO_ALUMNO AS
MEMBER FUNCTION NOTA RETURN NUMBER IS
BEGIN
        RETURN NOTA_FINAL;
END;
--
OVERRIDING MEMBER PROCEDURE VER_DATOS IS -- se redefine ese método
BEGIN
        DBMS_OUTPUT.PUT_LINE(CURSO|| '*'||NOTA_FINAL);
END;
END;
/
```

Mediante la cláusula **NOT FINAL** (incluida al final de la definición del tipo) se indica que se pueden derivar subtipos, si no se incluye esta cláusula se considera que es FINAL (no puede tener subtipos).

Igualmente si un método es **FINAL** los subtipos no pueden redefinirlo. La cláusula **OVERRIDING** se utiliza para redefinir el método. El siguiente bloque PL/SQL muestra un ejemplo de uso de los tipos definidos, al definir el objeto se inicializan todos los atributos ya que no se ha definido constructor para inicializar el objeto:

```
DECLARE
--Al asignar datos al alumno escribimos
-- DNI, NOMBRE, FECHA_NAC, CURSO, NOTA
        A1 TIPO_ALUMNO := TIPO_ALUMNO(NULL, NULL, NULL, NULL, NULL);
        A2 TIPO_ALUMNO := TIPO_ALUMNO('871234533A', 'PEDRO', '12/12/1996',
        'SEGUNDO', 7);

        NOM A1.NOMBRE%TYPE;
        DNI A1.DNI%TYPE;
        NOTAF A1.NOTA_FINAL%TYPE;
BEGIN
        A1.NOTA_FINAL:=8;
        A1.CURSO:='PRIMERO';
        A1.NOMBRE:='JUAN';
        A1.FEC_NAC:='20/10/1997';
        A1.VER_DATOS;
        NOM := A2.GET_NOMBRE();
        DNI := A2.GET_DNI();
        NOTAF := A2.NOTA();
        A2.VER_DATOS;
        DBMS_OUTPUT.PUT_LINE(A1.EDAD());
        DBMS_OUTPUT.PUT_LINE(A2.EDAD());
END;
/
```

A continuación se crea una tabla de TIPO_ALUMNO con el DNI como clave primaria, se insertan filas y se realiza alguna consulta (al insertar se escriben las columnas del supertipo - dni, nombre, fec_nac - y luego las del subtipo - curso, nota_final -):

```
CREATE TABLE TALUMNOS OF TIPO_ALUMNO (DNI PRIMARY KEY);

INSERT INTO TALUMNOS VALUES
        ('871234533A', 'PEDRO', '12/12/1996','SEGUNDO',7);
INSERT INTO TALUMNOS VALUES
        ('809004534B', 'MANUEL', '12/12/1997','TERCERO',8);

SELECT * FROM TALUMNOS;
SELECT DNI, NOMBRE, CURSO, NOTA_FINAL FROM TALUMNOS;
SELECT P.GET_DNI(), P.GET_NOMBRE(), P.EDAD(), P.NOTA() FROM TALUMNOS P;
```