Unidad Didáctica 4.

Bases de datos objeto-relacionales y orientadas a objetos.

Contenido

1.- INTRODUCCIÓN

Las bases de datos son fundamentales en muchos sistemas de información pero, las más tradicionales, son difíciles de utilizar cuando las aplicaciones que acceden a los datos utilizan lenguajes de programación orientados a objetos como C++ o Java. Este fue uno de los motivos de la creación de las bases de datos orientadas a objeto, además de dar solución a la aparición de aplicaciones más sofisticadas que necesitan tipos de datos y operaciones más complejas.

Los fabricantes de SGBD relacionales han ido incorporando en las nuevas versiones muchas de las propuestas para las bases de datos orientadas a objeto. Este es el caso de Informix, Oracle o PostgreSQL. Esto ha dado lugar al modelo relacional extendido y a los sistemas que lo implementan, que son los llamados **sistemas objeto-relacionales**.

2.- BASES DE DATOS OBJETO-RELACIONALES

Una **base de datos objeto-relacional** es una base de datos relacional que incorpora conceptos de la orientación a objetos. Por lo que un Sistema de Gestión de Bases de Datos Objeto-Relacional contiene las dos tecnologías. En este apartado vamos a estudiar la orientación a objeto que proporciona **Oracle**.

2.1 Características

Permite construir tipos de objetos complejos que tienen capacidad de definir objetos dentro de objetos y encapsulan o asocian métodos en dichos objetos.

Tipos de Objetos Oracle:

- Son tipos de datos definidos por el usuario.
- Pueden usarse como tipos de datos en columnas de tablas relacionales.
- Pueden usarse como tipo de datos en tablas de objetos
- Son una capa de abstracción sobre el modelo relacional de Oracle.
- Se pueden crear desde tipos de base de datos, cualquier tipo de objeto creado previamente, referencias de objetos y tipos colección.

Ventajas de los Objetos:

- El modelo de tipo de objetos es similar al mecanismo de clases en Java.
- Permiten por tanto un desarrollo más sencillo de entidades y lógica de negocio del mundo real.
- Mediante el soporte nativo de Oracle se habilita el acceso directo a las estructuras de datos.
- Permiten encapsular operaciones sobre los datos mediante métodos.

Características clave del modelo objeto-relacional:

- La interfaz tipo-objeto continua soportando la funcionalidad estándar de base de datos relacional como son:
- Herencia de tipo: tipos y subtipos.
- Evolución de tipo, mediante el comando ALTER TYPE para modificar sus características sin necesidad de recrearlo.

- Vistas de Objetos que permiten el desarrollo de aplicaciones orientadas a objetos sin necesidad de modificar el modelo relacional de la base de datos.
- Extensiones de Objeto SQL tanto en DML como DDL.
- Extensiones de Objeto PL/SQL para implementar la lógica y operaciones sobre tipos definidos por el usuario.
- Soporte Java para los Objetos Oracle mediante el uso de JDBC.

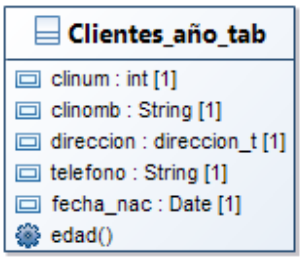
2.2 Tipos de objetos

Los tipos de objetos en Oracle son **tipos de datos definidos por el usuario**:

- Es una clase de tipo de dato.
- Podemos usarlos de la misma forma que usamos por ejemplo tipos de datos NUMBER y VARCHAR2.
- Un valor de tipo de objeto es una instancia de ese tipo.
- Una instancia de tipo objeto también se llama objeto.
- Diferencias con respecto a los tipos de datos nativos a una base de datos relacional:
  - No existe un conjunto de tipos de objeto predefinidos por lo que debemos definirlos nosotros mismos.
  - No son unitarios: tienen partes llamadas atributos y métodos.
    - Los **atributos** son características del objeto.
    - Los **métodos** son procedimientos o funciones que permiten hacer operaciones sobre los atributos.
- Los tipos de objetos son plantillas y los objetos son cosas que cumplen con dicha plantilla.

2.2.1 Definición del tipo de objeto

Para crear tipos de objetos se utiliza la sentencia **CREATE TYPE**. A continuación se muestran algunos ejemplos:




```
CREATE OR REPLACE TYPE direccion_t AS OBJECT (
        calle VARCHAR2(200),
        ciudad VARCHAR2(200),
        prov CHAR(20),
        codpos VARCHAR2(20));

CREATE OR REPLACE TYPE cliente_t AS OBJECT (
        clinum NUMBER,
        clinomb VARCHAR2(200),
        direccion direccion_t,
        telefono VARCHAR2(20),
        fecha_nac DATE,
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad,WNDS));
```

Crear un dato de tipo de objeto

Crear un dato de tipo de objeto, utilizando del definido anteriormente

Definir un método asociado al objeto

En Oracle, todos los tipos de objetos tienen asociado por defecto un constructor de nuevos objetos de ese tipo de acuerdo a la especificación del tipo. El nombre del método coincide con el nombre del tipo, y sus parámetros son los atributos del tipo. Por ejemplo las siguientes expresiones construyen dos objetos con todos sus valores.

```
direccion_t('Avenida Sagunto', 'Puzol', 'Valencia', 'E-23523')
cliente_t( 2347,'Juan Pérez Ruíz',direccion_t('Calle Eo', 'Onda', 'Castellón', '34568'),'696-779789','12/12/2012')
```

2.2.2. Métodos de los tipos de objetos

Normalmente, cuando se crea un objeto también se crean los métodos que permiten interactuar con él. Los métodos se especificarán después de los atributos del tipo.

2.3 Tablas de objetos

Una vez definidos los tipos, éstos pueden utilizarse para definir nuevos tipos, tablas que almacenen objetos de esos tipos, o para definir el tipo de los atributos de una tabla. **Una tabla de objetos es una clase especial de tabla que almacena un objeto en cada fila y que facilita el acceso a los atributos de esos objetos como si fueran columnas de la tabla**. Por ejemplo, se puede definir una tabla para almacenar los clientes de este año y otra para almacenar los de años anteriores de la siguiente manera:

```
CREATE TABLE clientes_año_tab OF cliente_t (clinum PRIMARY KEY);      -- Tabla de objetos

CREATE TABLE clientes_antiguos_tab (año NUMBER, cliente cliente_t);  -- Tabla que utiliza objetos en una columna
```

La diferencia entre la primera y la segunda tabla es que la primera almacena objetos con su propia identidad (**OID**) y la segunda no es una tabla de objetos, sino una tabla con una columna con un tipo de datos de objeto. Es decir, la segunda tabla tiene una columna con un tipo de datos complejo pero sin identidad de objeto. Además de esto, Oracle permite considerar una tabla de objetos desde dos puntos de vista:

- Como una tabla con una sola columna cuyo tipo es el de un tipo de objetos.
- Como una tabla que tiene tantas columnas como atributos los objetos que almacena.

Por ejemplo, se puede ejecutar una de las dos instrucciones siguientes. En la primera instrucción, la tabla **clientes_año_tab** se considera como una tabla con varias columnas cuyos valores son los especificados.

En el segundo caso, se la considera como con una tabla de objetos que en cada fila almacena un objeto. En esta instrucción la cláusula **VALUE** permite visualizar el valor de un objeto.

2.3.1. Inserción y selección de datos

```
INSERT INTO clientes_año_tab VALUES(
        2347,
        'Juan Perez Ruiz',
```

```
        direccion_t('Calle Castalia', 'Onda', 'Castellón', '34568'),
        '696-779789',
        '12/12/2012');

SELECT VALUE(c) FROM clientes_año_tab c
WHERE c.clinomb = 'Juan Perez Ruiz';
```

c es un alias

Las reglas de integridad, de clave primaria, y el resto de propiedades que se definan sobre una tabla, sólo afectan a los objetos de esa tabla, es decir no se refieren a todos los objetos del tipo asignado a la tabla.

Ejemplos

2.4 Referencia entre objetos

Para compartir objetos se utilizan referencias (**REFs** de forma abreviada). Una **REF** es un puntero al objeto en la tabla de objetos.

Los identificadores únicos (OID) asignados por Oracle a los objetos que se almacenan en una tabla, permiten que éstos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. El tipo de datos proporcionado por Oracle para soportar esta facilidad se denomina **REF**. **Un atributo de tipo REF almacena una referencia a un objeto del tipo definido, e implementa una relación de asociación entre los dos tipos de objetos.** Estas referencias se pueden utilizar para acceder a los objetos referenciados y para modificarlos; sin embargo, no es posible operar sobre ellas directamente. Para asignar o actualizar una referencia se debe utilizar siempre **REF** o **NULL**.

El siguiente ejemplo define un atributo de tipo **REF** y restringe su dominio a los objetos de cierta tabla.

```
CREATE TABLE clientes_tab OF cliente_t;

CREATE TYPE ordenes_t AS OBJECT (
        ordnum NUMBER,
        cliente REF cliente_t,
        fechpedido DATE,
        direcentrega direccion_t);

CREATE TABLE ordenes_tab OF ordenes_t (
        PRIMARY KEY (ordnum),
        SCOPE FOR (cliente) IS clientes_tab);
```

Cuando se borran objetos de la BD, puede ocurrir que otros objetos que referencien a los borrados queden en estado inconsistente. Estas referencias se denominan dangling references, y Oracle proporciona el predicado llamado **IS DANGLING** que permite comprobar cuándo sucede esto.

2.4.1. Inserción y acceso con referencias

La inserción de objetos con referencias implica la utilización del operador **REF** para poder insertar la referencia en el atributo adecuado. La siguiente sentencia inserta una orden de pedido en la tabla definida en la sección 2.4.

```
INSERT INTO ordenes_tab
SELECT 3001, REF(C),'30-MAY-1999', NULL
FROM cliente_tab C WHERE C.clinum= 3;
```

se seleccionan los valores de los 4 atributos de la tabla

El acceso a un objeto desde una referencia REF requiere primero referenciar al objeto. Para realizar esta operación, Oracle proporciona el operador **DEREF**. No obstante, utilizando la notación de punto también se consigue referenciar a un objeto de forma implícita.
Observemos el siguiente ejemplo:

```
CREATE TYPE persona_t AS OBJECT (
nombre VARCHAR2(30),
jefe REF persona_t ) ;
```

Si **x** es una variable que representa a un objeto de tipo **persona_t**, entonces las dos expresiones siguientes son equivalentes:

1. x.jefe.nombre
2. y.nombre, y=DEREF(x.jefe)

Para obtener una referencia a un objeto de una tabla de objetos, se puede aplicar el operador **REF** como muestra el siguiente ejemplo:

```
CREATE TABLE persona_tab OF persona_t;
DECLARE ref_persona REF persona_t;
SELECT REF(pe) INTO ref_persona FROM persona_tab pe WHERE pe.nombre= 'Juan Pérez Ruíz';
```

Simétricamente, para recuperar un objeto desde una referencia es necesario usar **DEREF**, como muestra el siguiente ejemplo que visualiza los datos del jefe de la persona indicada:

```
SELECT DEREF(pe.jefe) FROM persona_tab pe WHERE pe.nombre= 'Juan Pérez Ruíz';
```

En una base de datos con tipos y objetos, para evitar ambigüedades con los nombres de atributos y de métodos al utilizar la notación punto, **Oracle obliga a utilizar alias** para las tablas en la mayoría de las ocasiones, por eso lo más recomendable es utilizar siempre alias para los nombres de las tablas

2.5 Tipos de datos colección

Para poder implementar relaciones **1:N**, Oracle permite **definir tipos colección**. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera, es posible almacenar en un atributo un conjunto de tuplas en forma de array (**VARRAY**), o en forma de tabla anidada.

Al igual que los tipos objeto, los tipos colección también tienen por defecto unas funciones constructoras de colecciones cuyo nombre coincide con el del tipo. Los argumentos de entrada de estas funciones son el conjunto de elementos que forman la colección separados por comas y entre paréntesis, y el resultado es un valor del tipo colección.

En Oracle es posible diferenciar entre un valor nulo y una colección vacía. Para construir una colección sin elementos se puede utilizar la función constructora del tipo seguida por dos paréntesis sin elementos dentro.

2.5.1 El tipo VARRAY

Un array es un conjunto de elementos del mismo tipo. Cada elemento tiene asociado un índice que indica su posición dentro del array. Oracle permite que los **VARRAY** sean de longitud variable, aunque es necesario especificar un tamaño máximo cuando se declara el tipo **VARRAY**. Las siguientes declaraciones crean un tipo para una lista de precios, y un valor para dicho tipo.

```
CREATE TYPE precios AS VARRAY(10) OF NUMBER(12);
precios('35', '342', '3970');
```

Se puede utilizar el tipo **VARRAY** para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objeto.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

Cuando se declara un tipo **VARRAY** no se produce ninguna reserva de espacio. Si el espacio que requiere lo permite, se almacena junto con el resto de columnas de su tabla, pero si es demasiado largo (más de 4000 bytes) se almacena aparte de la tabla como un **BLOB**.

En el siguiente ejemplo, se define un tipo de datos para almacenar una lista de teléfonos. Este tipo se utiliza después para asignárselo a un atributo del tipo de objeto **cliente_t**.

```
CREATE TYPE lista_tel_t AS VARRAY(10) OF VARCHAR2(20) ;

CREATE TYPE cliente_t AS OBJECT (
        clinum NUMBER,
        clinomb VARCHAR2(200),
        direccion direccion_t,
        lista_tel lista_tel_t );
```

La principal limitación del tipo **VARRAY** es que **en las consultas es imposible poner condiciones sobre los elementos almacenados dentro**. Desde una consulta SQL, los valores de un **VARRAY** solamente pueden ser accedidos y recuperados como un bloque. Es decir, no se puede acceder individualmente a los elementos de un **VARRAY**. Sin embargo, desde un programa PL/SQL sí que es posible definir un bucle que itere sobre los elementos de un **VARRAY**.

2.5.2 Tablas anidadas

Una **tabla anidada** es un conjunto de elementos del mismo tipo sin ningún orden predefinido. Estas tablas solamente pueden tener una columna que puede ser de un tipo de datos básico de Oracle, o de un tipo de objeto definido por el usuario. En este último caso, la tabla anidada también puede ser considerada como una tabla con tantas columnas como atributos tenga el tipo de objeto. El siguiente ejemplo declara una tabla que después será anidada en el tipo **ordenes_t**. Los pasos de todo el diseño son los siguientes:

1. Se define el tipo de objeto **linea_t** para las filas de la tabla anidada.

```
CREATE TYPE linea_t AS OBJECT (
        linum NUMBER,
        item VARCHAR2(30),
        cantidad NUMBER,
        descuento NUMBER(6,2));
```

2. Se define el tipo colección tabla **lineas_pedido_t** para después anidarla.

```
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
```

Esta definición permite utilizar el tipo colección **lineas_pedido_t** para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objetos.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

3. Se define el tipo objeto **solicitudes_t** y su atributo **pedido** almacena una tabla anidada del tipo **lineas_pedido_t**.

```
CREATE TYPE solicitudes_t AS OBJECT (
        ordnum NUMBER,
        cliente REF cliente_t,
        fechpedido DATE,
        fechentrega DATE,
        pedido lineas_pedido_t,
        direcentrega direccion_t) ;
```

4. Se define la tabla de objetos **solicitudes_tab** y se especifica la tabla anidada del tipo **lineas_pedido_t.**

```
CREATE TABLE solicitudes_tab OF solicitudes_t
        (ordnum PRIMARY KEY,
        SCOPE FOR (cliente) IS clientes_tab)
        NESTED TABLE pedido STORE AS pedidos_tab ;
```

Este último paso es necesario realizarlo porque la declaración de una tabla anidada no reserva ningún espacio para su almacenamiento. Lo que se hace es indicar en qué tabla (**pedidos_tab**) se deben almacenar todas las líneas de pedido que se representen en el atributo pedido de cualquier objeto de la tabla **solicitudes_tab**. Es decir, todas las líneas de pedido de todas las **ordenes** se almacenan externamente a la tabla de **órdenes**, en otra tabla especial. Para relacionar las tuplas de una tabla anidada con la tupla a la que pertenecen, se utiliza una **columna oculta** que aparece en la tabla anidada por defecto. Todas las tuplas de una tabla anidada que pertenecen a la misma tupla tienen el mismo valor en esta columna (**NESTED_TABLE_ID**).

A diferencia de los **VARRAY**, los elementos de las tablas anidadas (**NESTED_TABLE**) sí pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos. En la próxima sección veremos cómo acceder individualmente a los elementos de una

tabla anidada mediante un cursor anidado. Además, las tablas anidadas pueden estar indexadas.

2.5.3 Inserción en tablas anidadas

Además del constructor del tipo de colección disponible por defecto, la inserción de elementos dentro de una tabla anidada puede hacerse siguiendo estas dos etapas:

1. Crear el objeto con la tabla anidada y dejar vacío el campo que contiene las tuplas anidadas.
2. Comenzar a insertar tuplas en la columna correspondiente de la tupla seleccionada por una subconsulta. Para ello, se tiene que utilizar la palabra clave **THE** con la siguiente sintaxis:

INSERT INTO THE (subconsulta) (tuplas a insertar)

Esta técnica es especialmente útil si dentro de una tabla anidada se guardan referencias a otros objetos. El siguiente ejemplo ilustra la manera de realizar estas operaciones sobre la tabla de ordenes (**ordenes_tab**) definida anteriormente.

```
CREATE TABLE clientes_tab OF cliente_t;

CREATE TYPE ordenes_t AS OBJECT (
        ordnum NUMBER,
        cliente REF cliente_t,
        fechpedido DATE,
        direcentrega direccion_t);

CREATE TABLE ordenes_tab OF ordenes_t (
        PRIMARY KEY (ordnum),
        SCOPE FOR (cliente) IS clientes_tab);
```

```
INSERT INTO solicitudes_tab
SELECT 3001, REF(C), SYSDATE,'30-MAY-1999', lineas_pedido_t(), NULL
FROM cliente_tab C
WHERE C.clinum= 3 ;
```

Inserta una orden

```
INSERT INTO THE (SELECT P.pedido FROM solicitudes_tab P WHERE P.ordnum = 3001 )
VALUES (linea_t(30, NULL, 18, 30));
```

Inserta una línea de pedido anidada

Para poner condiciones a las tuplas de una tabla anidada, se pueden utilizar cursores dentro de un **SELECT** o desde un programa PL/SQL. Veamos aquí un ejemplo de acceso con cursores. Utilizando el ejemplo de la sección 4.2.4, vamos a recuperar el número de las ordenes, sus fechas de pedido y las líneas de pedido que se refieran al item '**CH4P3**'.

```
SELECT ord.ordnum, ord.fechpedido,
        CURSOR (SELECT * FROM TABLE(ord.pedido) lp WHERE lp.item= 'CH4P3')
FROM solicitudes_tab ord;
```

La cláusula **THE** también sirve para seleccionar las tuplas de una tabla anidada. La sintaxis es como sigue:

SELECT ... **FROM THE** (subconsulta) **WHERE** ...

Por ejemplo, para seleccionar las primeras dos líneas de pedido de la orden **8778** se hace:

```
SELECT * FROM THE
        (SELECT ord.pedido FROM solicitudes_tab ord WHERE ord.ordnum= 8778) lp
WHERE lp.linum<3;
```

2.6 Métodos

Pueden ser de varios tipos:

- **MEMBER**: sirven para actuar con los objetos. Pueden ser procedimientos y funciones.
- **STATIC**: son independientes de las instancias del objeto. Pueden ser procedimientos y funciones.
- **CONSTRUCTOR**: sirve para inicializar el objeto. Es una función cuyos argumentos son los valores de los atributos del objeto y que devuelve el objeto inicializado.

No obstante podemos sobrescribir y/o crear otros constructores adicionales. Los constructores llevarán en la cláusula RETURN la expresión RETURN SELF AS RESULT.

La especificación de un método se hace junto a la creación de su tipo, y puede llevar asociada una directiva de compilación (PRAGMA RESTRICT_REFERENCES), para evitar que los métodos manipulen la base de datos o las variables del paquete PL/SQL. Tienen el siguiente significado:

- WNDS: no se permite al método modificar las tablas de la base de datos
- WNPS: no se permite al método modificar las variables del paquete PL/SQL
- RNDS: no se permite al método leer las tablas de la base de datos
- RNPS: no se permite al método leer las variables del paquete PL/SQL

Los métodos se pueden ejecutar sobre los objetos de su mismo tipo. Si x es una variable PL/SQL que almacena objetos del tipo Cliente_T, entonces x.edad() calcula la edad del cliente almacenado en x. La definición del cuerpo de un método en PL/SQL se hace de la siguiente manera:

En general, una vez creado el tipo con la especificación de los métodos se crea el cuerpo del nuevo tipo OBJECT mediante la instrucción CREATE OR REPLACE TYPE BODY:

A partir del dato fecha_nac calcular la edad

```
CREATE OR REPLACE TYPE BODY cliente_t AS
MEMBER FUNCTION edad RETURN NUMBER IS
    anio NUMBER;
    d DATE;
   BEGIN
    d:= sysdate;
    anio:= TO_NUMBER(TO_CHAR(d, 'YYYY')) - TO_NUMBER(TO_CHAR(fecha_nac, 'YYYY'));
    IF (TO_NUMBER(TO_CHAR(d, 'mm')) < TO_NUMBER(TO_CHAR(fecha_nac, 'mm'))) OR
      (TO_NUMBER(TO_CHAR(d, 'mm')) = TO_NUMBER(TO_CHAR(fecha_nac, 'mm')))
        AND (TO_NUMBER(TO_CHAR(d, 'dd')) < TO_NUMBER(TO_CHAR(fecha_nac, 'dd')))
    THEN anio:= anio-1;
    END IF;
    RETURN anio;
   END;
END;
```

Para eliminarlo:

DROP TYPE BODY nombre_tipo:

2.6.1 Sobrecarga

Los métodos del mismo tipo (funciones y procedimientos) se pueden sobrecargar, esto es, es posible utilizar el mismo nombre para métodos distintos si sus parámetros formales difieren en número, orden o tipo de datos.

2.6.2 Métodos MAP y ORDER

Los valores de un tipo escalar, como CHAR o REAL, tienen un orden predefinido que permite compararlos.

Sin embargo, las instancias de un objeto carecen de un orden predefinido. **Para comparar los objetos de cierto tipo** es necesario indicar a Oracle cuál es el criterio de comparación. Para ello, hay que escoger entre **un método MAP u ORDER**, debiéndose definir uno de estos métodos por cada tipo de objeto que necesite ser comparado. La diferencia entre ambos es la siguiente:

- Un método **MAP** sirve para **indicar cuál de los atributos del tipo se utilizará para ordenar los objetos del tipo**, y por tanto se puede utilizar para comparar los objetos de ese tipo por medio de los **operadores de comparación** (**<, >**).

  Un tipo de objeto puede contener sólo una función de MAP, que necesariamente debe **carecer de parámetros** y debe devolver uno de los siguientes tipos escalares: DATE, NUMBER, VARCHAR2 y cualquiera de los tipos ANSI SQL (como CHARACTER o REAL).

  Por ejemplo, la siguiente declaración permite decir que los objetos del tipo **cliente_t** se van a comparar por su atributo clinum.

```
CREATE TYPE cliente_t AS OBJECT (
clinum NUMBER,
clinomb VARCHAR2(200),
direccion direccion_t,
telefono VARCHAR2(20),
fecha_nac DATE,
MAP MEMBER FUNCTION ret_value RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(ret_value, WNDS, WNPS, RNPS, RNDS),
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad, WNDS));

CREATE OR REPLACE TYPE BODY cliente_t AS
MAP MEMBER FUNCTION ret_value RETURN NUMBER IS
        BEGIN
        RETURN clinum
        END;
END;
```

- 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.

Un tipo de objeto puede contener un único método ORDER, que es una función que devuelve un resultado numérico.

Este método devolverá un valor negativo si el parámetro de entrada es mayor que el atributo, un valor positivo si ocurre lo contrario y un cero si ambos son iguales. El siguiente ejemplo define un orden para el tipo **cliente_t** diferente al anterior. Sólo una de estas definiciones puede ser válida en un tiempo dado.

```
CREATE TYPE cliente_t AS OBJECT (
        clinum NUMBER,
        clinomb VARCHAR2(200),
        direccion direccion_t,
        telefono VARCHAR2(20),
        fecha_nac DATE,
ORDER MEMBER FUNCTION cli_ordenados (x IN cliente_t) RETURN INTEGER,
PRAGMA RESTRICT_REFERENCES (cli_ordenados, WNDS, WNPS, RNPS, RNDS),
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad, WNDS));

CREATE OR REPLACE TYPE BODY cliente_t AS
ORDER MEMBER FUNCTION cli_ordenados (x IN cliente_t)
RETURN INTEGER IS
        BEGIN
        RETURN clinum - x.clinum; /*la resta de los dos números clinum*/
        END;
END;
```

Si un tipo de objeto no tiene definido ninguno de estos métodos, Oracle es incapaz de deducir cuándo un objeto es mayor o menor que otro. Sin embargo, sí puede determinar cuándo dos objetos del mismo tipo son iguales. Para ello, el sistema compara el valor de los atributos de los objetos uno a uno:

- Si todos los atributos son no nulos e iguales, Oracle indica que ambos objetos son iguales.
- Si alguno de los atributos no nulos es distinto en los dos objetos, entonces Oracle dice que son diferentes.
- En otro caso, Oracle dice que no puede comparar ambos objetos.

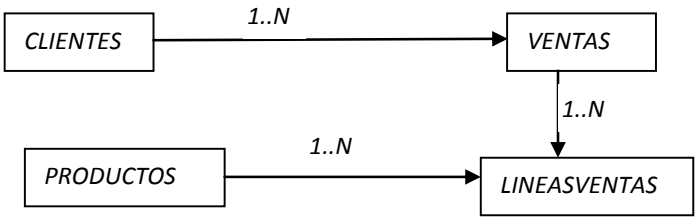
Es importante tener en cuenta los siguientes puntos:

- Un método MAP proyecta el valor de los objetos en valores escalares (que son más fáciles de comparar).
  Si un tipo de objeto define uno de estos métodos, el método se llama automáticamente para evaluar comparaciones del tipo obj1 > obj2 y para evaluar las comparaciones que implican DISTINCT, GROUP BY y ORDER BY.
  Un método ORDER simplemente compara el valor de un objeto con otro.
- Se puede declarar un método MAP o un método ORDER, pero no ambos.
- Si se declara uno de los dos métodos, es posible comparar objetos en SQL o en un procedimiento. Sin embargo, si no se declara ninguno, sólo se pueden comparar la igualdad o desigualdad de dos objetos y sólo en SQL.

Cuando es necesario ordenar un número grande de objetos es mejor utilizar un método MAP (ya que una llamada por objeto proporciona una proyección escalar que es más fácil de ordenar). Un método ORDER es menos eficiente: debe invocarse repetidamente ya que compara sólo dos objetos cada vez.

3.- EJEMPLO DE TRANSFORMACIÓN DEL MODELO RELACIONAL

CLIENTES(idcliente, calle, población, codpostal, provincia, nif, telef1, telef2, telef3)
PRODUCTOS(idproducto, descripción, pvp, stockactual)
VENTAS(idventa, idcliente, fechaventa)
LINEASVENTAS(idventa,numerolinea, idproducto, cantidad)



A. Implementación como base de datos relacional

```
CREATE TABLE CLIENTES (
Idcliente NUMBER PRIMARY KEY,
calle VARCHAR2(50),
poblacion VARCHAR2(50),
codpostal VARCHAR2(5),
provincia VARCHAR2(50),
nif VARCHAR2(9) UNIQUE,
telef1 VARCHAR2(15),
telef2 VARCHAR2(15),
telel3 VARCHAR2(15)
);

CREATE TABLE PRODUCTOS (
idproducto NUMBER PRIMARY KEY,
descripcion VARCHAR2(80),
pvp NUMBER,
stockactual NUMBER
);

CREATE TABLE VENTAS (
idventa NUMBER PRIMARY KEY,
idcliente NUMBER NOT NULL REFERENCES CLIENTES,
fechaventa DATE
);

CREATE TABLE LINEASVENTAS (
idventa NUMBER,
numerolinea NUMBER,
idproducto NUMBER,
cantidad NUMBER,
foreign KEY (idventa) references VENTAS (idventa),
foreign key (idproducto) references PRODUCTOS(idproducto),
primary key (idventa, numerolinea)
);
```

B. Implementación como base de datos objeto-relacional

1. Definir un tipo VARRAY para contener los teléfonos:

```
CREATE TYPE TIP_TELEFONOS AS VARRAY(3) OF VARCHAR2(15);
```

2. Crear los tipos direccion, cliente, producto y linea de venta:

```
CREATE TYPE TIP_DIRECCION AS OBJECT (
CALLE VARCHAR2(50),
POBLACION VARCHAR2(50),
CODPOSTAL NUMBER(5),
PROVINCIA VARCHAR2(20));

CREATE TYPE TIP_CLIENTE AS OBJECT (
IDCLIENTE NUMBER,
NOMBRE VARCHAR2(50),
DIREC TIP_DIRECCION,
NIF VARCHAR2(9),
TELEF TIP_TELEFONOS);

CREATE TYPE TIP_PRODUCTO AS OBJECT(
IDPRODUCTO NUMBER,
DESCRIPCION VARCHAR2(80),
PVP NUMBER,
STOCKACTUAL NUMBER);

CREATE TYPE TIP_LINEAVENTA AS OBJECT(
NUMEROLINEA NUMBER,
IDPRODUCTO REF TIP_PRODUCTO,
CANTIDAD NUMBER);
```

3. Crear un tipo tabla anidada para contener las líneas de una venta:

```
CREATE TYPE TIP_LINEAS_VENTA AS TABLE OF TIP_LINEAVENTA;
```

4. Crear un tipo venta para los datos de las ventas, cada venta tendrá un atributo LINEAS del tipo tabla anidada. Se define además la función TOTAL_VENTA que calcula la venta de la venta de las líneas de venta que forman parte de una venta.

```
CREATE TYPE TIP_VENTA AS OBJECT(
IDVENTA NUMBER,
IDCLIENTE REF TIP_CLIENTE,
FECHAVENTA DATE,
LINEAS TIP_LINEAS_VENTA,
MEMBER FUNCTION TOTAL_VENTA RETURN NUMBER);

CREATE OR REPLACE TYPE BODY TIP_VENTA AS
  MEMBER FUNCTION TOTAL_VENTA RETURN NUMBER IS
    TOTAL NUMBER:=0;
    LINEA TIP_LINEAVENTA;
    PRODUCT TIP_PRODUCTO;
  BEGIN
```

```
    FOR I IN 1..LINEAS.COUNT LOOP
     LINEA:=LINEAS(I);
     SELECT DEREF(LINEA.IDPRODUCTO) INTO PRODUCT FROM DUAL;
     TOTAL:=TOTAL+LINEA.CANTIDAD*PRODUCT.PVP;
    END LOOP;
    RETURN TOTAL;
  END;
END;
```

5. Crear las tablas para almacenar los objetos de la aplicación.

```
CREATE TABLE TABLA_CLIENTES OF TIP_CLIENTE(
IDCLIENTE PRIMARY KEY,
NIF UNIQUE);

CREATE TABLE TABLA_PRODUCTOS OF TIP_PRODUCTO(
IDPRODUCTO PRIMARY KEY);

CREATE TABLE TABLA_VENTAS OF TIP_VENTA(
IDVENTA PRIMARY KEY)
NESTED TABLE LINEAS STORE AS TABLA_LINEAS;
```

6. Insertar datos en clientes y productos

```
INSERT INTO TABLA_PRODUCTOS VALUES (1, 'CAJA DE CRISTAL MURANO', 100,5);
INSERT INTO TABLA_PRODUCTOS VALUES (2,'BICICLETA CITY',120,15);
INSERT INTO TABLA_PRODUCTOS VALUES (3, '100 LAPICES DE COLORES', 20,5);
INSERT INTO TABLA_PRODUCTOS VALUES (4,'OPERACIONES CON BD',25,5);
INSERT INTO TABLA_PRODUCTOS VALUES (5,'APLICACIONES WEB',25.50,10);

INSERT INTO TABLA_CLIENTES VALUES
 (1,'Luis Garcia',TIP_DIRECCION('C/ Las Flores 23','Guadalajara','19003','Guadalajara'),
 '12345678L',TIP_TELEFONOS('976123456','69876544'));

INSERT INTO TABLA_CLIENTES VALUES
 (2,'Ana Serrano',TIP_DIRECCION('C/ Galiana 2','Guadalajara','19004','Guadalajara'),
 '98765432L',TIP_TELEFONOS('976123587','69874584'));
```

7. Insertar en TABLA_VENTAS una venta para el IDCLIENTE 1:

```
INSERT INTO TABLA_VENTAS
SELECT 1, REF(C),SYSDATE,TIP_LINEAS_VENTA()
FROM TABLA_CLIENTES C WHERE C.IDCLIENTE=1;
```

8. Insertar en TABLA_VENTAS dos líneas de venta para el IDVENTA 1 para los productos 1 (la CANTIDAD es 1) Y 2 (la CANTIDAD es 2):

```
INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=1)
(SELECT 1,REF(P), 1 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=1);

INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=1)
```

```
(SELECT 2,REF(P), 2 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=2);
```

9. Insertar en TABLA_VENTAS una venta con IDVENTA 2 para IDCLIENTE1, y posteriormente introducir líneas de venta

```
INSERT INTO TABLA_VENTAS
SELECT 2, REF(C),SYSDATE,TIP_LINEAS_VENTA()
FROM TABLA_CLIENTES C WHERE C.IDCLIENTE=1;

INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=2)
(SELECT 1,REF(P), 2 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=1);
INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=2)
(SELECT 2,REF(P), 1 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=4);
INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=2)
(SELECT 3,REF(P), 4 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=5);
```

10. Procedimiento para visualizar los datos de la venta:

```
 CREATE OR REPLACE PROCEDURE VER_VENTA (ID NUMBER) AS
 LIN NUMBER;
 CANT NUMBER;
 IMPORTE NUMBER;
 TOTAL_V NUMBER;
 PRODUC TIP_PRODUCTO:=TIP_PRODUCTO(NULL,NULL,NULL,NULL);
 CLI TIP_CLIENTE:=TIP_CLIENTE(NULL,NULL,NULL,NULL,NULL);
 DIR TIP_DIRECCION:=TIP_DIRECCION(NULL,NULL,NULL,NULL);
 FEC DATE;
 CURSOR C1 IS
  SELECT NUMEROLINEA, DEREF(IDPRODUCTO), CANTIDAD FROM THE
   (SELECT T.LINEAS FROM TABLA_VENTAS T WHERE IDVENTA=ID);
 BEGIN
  SELECT DEREF(IDCLIENTE),FECHAVENTA,V.TOTAL_VENTA()
  INTO CLI, FEC, TOTAL_V
  FROM TABLA_VENTAS V WHERE IDVENTA=ID;
  DIR:=CLI.DIREC;
  DBMS_OUTPUT.PUT_LINE('NUMERO DE VENTA: '||ID||'*Fecha de venta: '||FEC);
  DBMS_OUTPUT.PUT_LINE('CLIENTE: '||CLI.NOMBRE);
  DBMS_OUTPUT.PUT_LINE('DIRECCION: '||DIR.CALLE);

DBMS_OUTPUT.PUT_LINE('================================================='
);

  OPEN C1;
  FETCH C1 INTO LIN, PRODUC, CANT;
  WHILE C1%FOUND LOOP
   IMPORTE:=CANT*PRODUC.PVP;

DBMS_OUTPUT.PUT_LINE(LIN||'*'||PRODUC.DESCRIPCION||'*'||PRODUC.PVP||'*'||
CANT||'*'||IMPORTE);
    FETCH C1 INTO LIN,PRODUC,CANT;
  END LOOP;
```

```
  CLOSE C1;
  DBMS_OUTPUT.PUT_LINE('Total venta: '||TOTAL_V);
END VER_VENTA;
```

Para ejecutar el procedimiento

```
SET serveroutput ON format wrapped;

BEGIN
VER_VENTA(2);
END;
```