

Unidad 7.- Bases de Datos Objeto-Relacionales

1. Introducción

Las estructuras de datos que se utilizan para almacenar la información siguen siendo tablas, los usuarios pueden utilizar muchos de los mecanismos de orientación a objetos para definir y acceder a los datos. Por esta razón, se dice que se trata de un modelo de datos objeto-relacional.

Oracle proporciona mecanismos para que el usuario pueda definir sus propios tipos de datos, cuya estructura puede ser compleja, y que se pueden aplicar para asignar un tipo a una columna de una tabla. También reconoce el concepto de objetos, de tal manera que un objeto tiene un tipo, se almacena en cierta fila de cierta tabla y tiene un identificador único (OID). Estos identificadores se pueden utilizar para referenciar a otros objetos y así representar relaciones de asociación y de agregación. Oracle también proporciona mecanismos para asociar métodos a tipos, y constructores para diseñar tipos de datos multivaluados (colecciones) y tablas anidadas. La mayor deficiencia de este sistema es la imposibilidad de definir jerarquías de especialización y herencia, lo cual es una importante desventaja con respecto a las bases de datos orientadas a objetos.

2. Tipos de Datos Definidos por el Usuario

Los usuarios de Oracle pueden definir sus propios tipos de datos, pudiendo ser de dos categorías: tipos de objetos (*object types*) y tipos para colecciones (*collection types*). Para construir los tipos de usuario se utilizan los tipos básicos provistos por el sistema y otros tipos de usuario previamente definidos. Un tipo define una estructura y un comportamiento común para un conjunto de datos de las aplicaciones.

2.1 Tipos de objetos

Un tipo de objetos representa a una entidad del mundo real. Un tipo de objetos se compone de los siguientes elementos:

- Un nombre que sirve para identificar el tipo de los objetos.
- Unos atributos que modelan la estructura y los valores de los datos de ese tipo. Cada atributo puede ser de un tipo de datos básico o de un tipo de usuario.
- Unos métodos que son procedimientos o funciones escritos en el lenguaje PL/SQL (almacenados en la base de datos), o escritos en C (almacenados externamente).

Los tipos de objetos pueden interpretarse como plantillas a las que se adaptan los objetos de ese tipo. A continuación se da un ejemplo de cómo definir el tipo de datos `Direccion_T` en el lenguaje de definición de datos de Oracle, y como utilizar este tipo de datos para definir el tipo de datos de los objetos de la clase de `Clientes_T`.

DEFINICIÓN ORIENTADA A OBJETOS

define type `Direccion_T`:

tuple [`calle:string`,
`ciudad:string`,
`prov:string`,
`codpos:string`]

define class `Clientes_T`

type tuple [`clinum: integer`,
`clinomb:string`,
`direccion:Direccion_T`,
`telefono: string`, `fecha-nac:date`]

operations `edad():integer`

DEFINICIÓN EN ORACLE

CREATE TYPE `direccion_t` **AS OBJECT** (

`calle VARCHAR2(200),`
`ciudad VARCHAR2(200),`
`prov CHAR(2),`
`codpos VARCHAR2(20)`) ;

CREATE TYPE `clientes_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)`
) ;

2.2 Métodos

La especificación de un método se hace junto con la creación de su tipo, y debe llevar siempre asociada una directiva de compilación (`PRAGMA RESTRICT_REFERENCES`):

- 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 `Clientes_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:

```

CREATE OR REPLACE TYPE BODY clientes_t AS
  MEMBER FUNCTION edad RETURN NUMBER IS
    a      NUMBER;
    d      DATE;
  BEGIN
    d:= today();
    a:= d.año - fecha_nac.año; IF
      (d.mes < fecha_nac.mes) OR
      ((d.mes = fecha_nac.mes) AND (d.dia <
        fecha_nac.dia)) THEN a:= a+1;
    END IF;
    RETURN a;
  END;
END;

```

2.2.1 Constructores de tipo

En Oracle, todos los tipos de objetos tienen asociado por defecto un método que construye 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')
clientes_t( 2347,
            'José Pérez Ruíz',
            direccion_t('Calle Eo', 'Onda', 'Castellón', '34568'),
            '696-779789',
            12/12/1981
            )

```

2.2.2 Métodos de comparación

Para comparar los objetos de cierto tipo es necesario indicar a Oracle cuál es el criterio de comparación. Para hacer esto hay que escoger entre un método MAP o ORDER, debiéndose definir al menos uno de estos métodos por cada tipo de objetos que necesiten ser comparados. La diferencia entre ellos es la siguiente:

- Un método de MAP sirve para indicar cuál de los atributos del tipo se va a utilizar para ordenar los objetos del tipo, y por lo tanto se puede utilizar para comparar los objetos de ese tipo por medio de los operadores de comparación típicos (<, >). Por ejemplo la siguiente declaración permite decir que los objetos del tipo clientes_t se van a comparar por su atributo clinum.

```

CREATE TYPE clientes_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), /*instrucciones a PL/SQL*/

```

```

MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES (edad, WNDS)
) ;

CREATE OR REPLACE TYPE BODY clientes_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. Este método debe devolver un valor negativo si el primero es mayor que el segundo, un valor positivo si ocurre lo contrario y un cero si ambos son iguales. El siguiente ejemplo define un orden para el tipo `clientes_t` diferente al anterior. Solo una de estas definiciones puede ser válida a un tiempo.

```

CREATE TYPE clientes_t AS OBJECT
    ( clinum NUMBER,
      clinomb VARCHAR2(200),
      direccion direccion_t,
      telefono VARCHAR2(20),
      fecha_nac DATE,

ORDER MEMBER FUNCTION
      cli_ordenados (x IN clientes_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 clientes_t AS
    ORDER MEMBER FUNCTION cli_ordenados (x IN clientes_t)
        RETURN INTEGER IS
        BEGIN
            RETURN clinum - x.clinum; /*la resta de los dos números clinum*/
        END;
END;

```

Para un tipo de objetos que no tenga definido ninguno de estos métodos, Oracle es incapaz de deducir cuándo un objeto es mayor o menor que otro. Sin embargo sí que puede determinar cuándo dos objetos del mismo tipo son iguales. Para hacer esto, 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.

2.3 Tablas de objetos

Después de definir los tipos, éstos pueden utilizarse para definir otros 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
  clientes_t (clinum PRIMARY KEY);

CREATE TABLE clientes_antiguos_tab (
  año NUMBER,
  cliente clientes_t
) ;
```

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 objetos. 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.

```
INSERT INTO clientes_año_tab
VALUES ( 2347,
  'José Pérez Ruíz',
  direccion_t('Calle Castalia', 'Onda', 'Castellón',
  '34568'), '696-779789', 12/12/1981
) ;

SELECT VALUE(c) FROM clientes_año_tab c
WHERE c.clinomb = 'José Pérez Ruíz'
```

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.

2.4 Referencias entre objetos

Los identificadores únicos 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.

Cuando se define una columna de un tipo a REF, es posible restringir su dominio a los objetos que se almacenen en cierta tabla. Si la referencia no se asocia a una tabla sino que sólo se restringe a un tipo de objetos, se podrá actualizar a una referencia a un objeto del tipo adecuado independientemente de la tabla donde se almacene. En este caso su almacenamiento requerirá más espacio y su acceso será menos eficiente. El siguiente ejemplo define un atributo de tipo REF y restringe su dominio a los objetos de cierta tabla.

```
CREATE TABLE clientes_tab OF clientes_t;

CREATE TYPE ordenes_t AS OBJECT (
    ordnum NUMBER, cliente REF
    clientes_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 base de datos, puede ocurrir que otros objetos que referencien a los borrados queden en un estado inconsistente. Estas referencias se denominan *dangling references*, y Oracle proporciona un predicado que permite comprobar cuando sucede esto. El predicado se denomina IS DANGLING.

2.5 Tipos para colecciones

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

Al igual que los tipos para objetos, los tipos para colecciones 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 ordenado 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 ordenada de precios, y un valor para dicho tipo.

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

Un tipo VARRAY se puede utilizar para:

- Definir el tipo de datos de una columna de una tabla relacional.
- Definir el tipo de datos de un atributo de un tipo de objetos.
- 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 quiere definir un tipo de datos para almacenar una lista ordenada de teléfonos (tipo list, en el tipo set no existe orden). Este tipo se utiliza después para asignárselo a un atributo del tipo de objetos `clientes_t`.

DEFINICIÓN ORIENTADA A OBJETOS

```
define type Lista_Tel_T: list(string);
```

```
define class Clientes_T:
    tuple [clinum: integer,
           clinomb:string,
           direccion:Direccion_T,
           lista_tel: Lista_Tel_T];
```

DEFINICIÓN EN ORACLE

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

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

La principal limitación que presenta los tipos VARRAY es que en las consultas es imposible poner condiciones sobre los elementos almacenados dentro. Desde una consulta SQL, los valores de un VARRAY sólo pueden ser accedidos y recuperados en un bloque. Es decir no se puede acceder a los elementos de un VARRAY individualmente. Sin embargo desde un programa PL/SQL sí que es posible definir un bucle que itere sobre los elementos de un VARRAY (ver sección 4.2.5).

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 objetos 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 objetos. En el siguiente ejemplo, se declara una tabla que después es anidada en el tipo `ordenes_t`. Los pasos de todo el diseño son los siguientes.

1- Se define el tipo de objetos `linea_t` para las filas de la tabla anidada.

```
define type Linea_T:
    tuple [linum:integer,
           item:string,
           cantidad:integer,
           descuento:real];

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

2- Se define el tipo de 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 datos de una columna de una tabla relacional.
- Definir el tipo de datos 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 de objetos `ordenes_t` que en el atributo `pedido` almacena una tabla anidada del tipo `lineas_pedido_t`.

```
define class Ordenes_T:
    tuple [ordnum:integer,
           cliente:Clientes_T,
           fechpedido:date,
           fechentrega:date,
           pedido:set(Linea_T),
           direcentrega:Direccion_T];

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

4- Se define la tabla de objetos `ordenes_tab` y se especifica la tabla anidada del tipo `lineas_pedido_t`.

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

Este último paso es necesario hacerlo 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 `ordenes_tab`. Es decir, todas las líneas de pedido de todas las ordenes se almacenan externamente a la tabla de ordenes, 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`).

Al contrario que los VARRAY, los elementos de las tablas anidadas si pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos. Como veremos en la próxima sección, una forma conveniente de acceder individualmente a los elementos de una tabla anidada es por medio de un cursor anidado. Además, las tablas anidadas pueden ser indexadas.

3. Inserción y Acceso a los Datos

3.1 Alias

En una base de datos con tipos y objetos, lo más recomendable es utilizar siempre alias para los nombres de las tablas. El alias de una tabla debe ser único en el contexto de una consulta. Los alias se utilizan para acceder al contenido de las tablas, pero hay que tener cuidado de utilizarlos adecuadamente en las tablas que almacenan objetos. El siguiente ejemplo ilustra cómo se deben utilizar.

```
CREATE TYPE persona AS OBJECT (nombre VARCHAR(20));
CREATE TABLE ptab1 OF persona;
CREATE TABLE ptab2 (c1 persona);
CREATE TABLE ptab3 (c1 REF persona);
```

La diferencia entre las dos tablas está en que la primera almacena objetos del tipo `persona`, mientras que la segunda tabla tiene una columna donde se almacenan valores del tipo `persona`. Considerando ahora las siguientes consultas, se ve cómo se puede acceder a estas tablas.

1. <code>SELECT nombre FROM ptab1;</code>	BIEN
2. <code>SELECT c1.nombre FROM ptab2;</code>	MAL
3. <code>SELECT p.c1.nombre FROM ptab2 p;</code>	BIEN
4. <code>SELECT p.c1.nombre FROM ptab3 p;</code>	BIEN
5. <code>SELECT p.nombre FROM ptab3 p;</code>	MAL

En la primera consulta `nombre` es considerado como una de las columnas de la tabla `ptab1`, ya que los atributos de los objetos se consideran columnas de la tabla de objetos. Sin embargo, en la segunda consulta se requiere la utilización de un alias para indicar que `nombre` es el nombre de un atributo del objeto de tipo `persona` que se almacena en la columna `c1`. Para resolver este problema no es posible utilizar los nombres de las tablas directamente: `ptab2.c1.nombre` es incorrecto. Las consultas 4 y 5 muestran cómo acceder a los atributos de los objetos referenciados desde un atributo de la tabla `ptab3`.

En conclusión, para facilitar la formulación de consultas y evitar errores se recomienda utilizar alias para acceder a todas las tablas que contengan objetos con o sin identidad, y para acceder a las columnas de las tablas en general.

3.2 Inserción de referencias

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

```
INSERT INTO ordenes_tab
  SELECT 3001, REF(C), '30-MAY-1999', NULL
  --se seleccionan los valores de los 4 atributos de la tabla
  FROM clientes_tab C WHERE C.clinum= 3 ;
```

El acceso a un objeto desde una referencia REF requiere dereferenciar al objeto primero. Para realizar esta operación, Oracle proporciona el operador Deref. No obstante, utilizando la notación de punto también se consigue dereferenciar 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 de la manera que se muestra en 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= 'José Pérez Ruíz';
```

Simétricamente, para recuperar un objeto desde una referencia es necesario usar Deref, como se muestra en este ejemplo que visualiza los datos del jefe de la persona indicada:

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

3.3 Llamadas a métodos

Para invocar un método hay que utilizar su nombre y unos paréntesis que encierren sus argumentos de entrada. Si el método no tienen argumentos hay que especificar los paréntesis aunque estén vacíos. Por ejemplo, si *tb* es una tabla con la columna *c* de tipo de objetos *t*, y *t* tiene un método *m* sin argumentos de entrada, la siguiente consulta es correcta:

```
SELECT p.c.m( ) FROM tb p;
```

3.4 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 el campo que contiene las tuplas anidadas vacío.
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 cuando dentro de una tabla anidada se guardan referencias a otros objetos. En el siguiente ejemplo se ilustra la manera de realizar estas operaciones sobre la tabla de ordenes (*ordenes_tab*) definida en la sección 2.5.2.

```
INSERT INTO ordenes_tab          --inserta una orden
  SELECT 3001, REF(C),
         SYSDATE, '30-MAY-1999',
         lineas_pedido_t(),
         NULL
  FROM   clientes_tab C
  WHERE  C.clinum= 3 ;

INSERT INTO THE (                --selecciona el atributo pedido de la orden
  SELECT P.pedido
  FROM   ordenes_tab P
  WHERE  P.ordnum = 3001
)
  SELECT 30, REF(S), 18, 30 --inserta una linea de pedido anidada
  FROM   items_tab S
  WHERE  S.itemnum = 3011;
```

Para poner condiciones sobre las tuplas de una tabla anidada, se pueden utilizar cursores dentro de un **SELECT** o desde un programa PL/SQL de la manera explicada en la sección 4.2.5. Veamos aquí un ejemplo de acceso con cursores. Utilizando el ejemplo de la sección 2.5.2, vamos a recuperar el número de las ordenes, sus fechas de pedido y las líneas de pedido que se refieran al ítem 'CH4P3'.

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

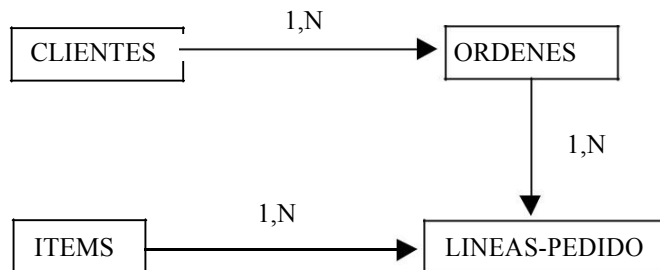
```
SELECT lp FROM THE
(SELECT ord.pedido FROM ordenes_tab ord WHERE ord.ordnum= 8778) lp
WHERE lp.linum<3;
```

4. Una Base de Datos Ejemplo

Partiendo de una base de datos para gestionar los pedidos de los clientes, veamos como se puede proporcionar una solución relacional y otra objeto_relacional en Oracle.

4.1 Modelo lógico para una base de datos relacional

CLIENTES(clinum, clinomb, calle, ciudad, prov, codpos, tel1, tel2, tel3)
ORDENES(ordnum, clinum, fechpedido, fechentrega, callent, ciuent, provent, codpent)
ITEMS(numitem, precio, tasas)
LINEAS(linum, ordnum, numitem, cantidad, descuento)



4.1.1 Implementación relacional con Oracle

Se crean tablas normalizadas y con claves ajenas para representar las relaciones.

```
CREATE TABLE clientes (
    clinum NUMBER,
    clinomb VARCHAR2(200),
    calle VARCHAR2(200),
    ciudad VARCHAR2(200),
    prov CHAR(2),
    codpos VARCHAR2(20),
    tel1 VARCHAR2(20),
    tel2 VARCHAR2(20),
    tel3 VARCHAR2(20),
    PRIMARY KEY (clinum)
);

CREATE TABLE ordenes (
    ordnum NUMBER,
    clinum NUMBER REFERENCES clientes,
    fechpedido DATE,
    fechaentrega DATE,
    callent VARCHAR2(200),
    ciuent VARCHAR2(200),
    provent CHAR(2),
```

```

        codpent      VARCHAR2(20),
        PRIMARY KEY (ordnum)
    ) ;

CREATE TABLE items (
    numitem NUMBER PRIMARY KEY,
    precio NUMBER,
    tasas    NUMBER
) ;

CREATE TABLE lineas (
    linum      NUMBER,
    ordnum     NUMBER REFERENCES ordenes,
    numitem    NUMBER REFERENCES items,
    cantidad   NUMBER,
    descuento  NUMBER,
    PRIMARY KEY (ordnum, linum)
) ;

```

4.2 Modelo lógico para una base de datos orientada a objetos

Primero vamos utilizar el lenguaje de definición de bases de datos orientadas a objetos visto en el tema 5 para definir el esquema de la base de datos que después crearemos en Oracle.

```

define type Lista_Tel_T: type list(string);

define type Direccion_T: type tuple [ calle:string,
                                     ciudad:string,
                                     prov:string,
                                     codpos:string];

define class Clientes_T: type tuple [ clinum: integer,
                                     clinomb:string,
                                     direccion:Direccion_T,
                                     lista_tel: Lista_Tel_T];

define class Item_T: type tuple [ itemnum:integer,
                                precio:real,
                                tasas:real];

define type Linea_T: type tuple [linum:integer,
                                item:Item_T,
                                cantidad:integer,
                                descuento:real];

define type Lineas_Pedido_T: type set(Linea_T);

define class Ordenes_T: type tuple [ ordnum:integer,
                                    cliente:Clientes_T,
                                    fechpedido:date,
                                    fechentrega:date,
                                    pedido:Lineas_Pedido_T
                                    direcentrega:Direccion_T];

```

4.2.1 Implementación objeto-relacional con Oracle

Aquí se indica como definir todos los tipos anteriores en Oracle.

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

CREATE TYPE direccion_t AS OBJECT (
    calle  VARCHAR2(200),
    ciudad VARCHAR2(200),
    prov   CHAR(2),
    codpos VARCHAR2(20)
) ;

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

CREATE TYPE item_t AS OBJECT
( itemnum NUMBER,
  precio NUMBER,
  tasas NUMBER
) ;

CREATE TYPE linea_t AS OBJECT
( linum NUMBER,
  item REF item_t,
  cantidad NUMBER,
  descuento NUMBER
) ;

CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;

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

4.2.2 Creación de tablas de objetos

Ahora vamos a crear las tablas donde almacenar los objetos de la aplicación.

```
CREATE TABLE clientes_tab OF
  clientes_t (clinum PRIMARY KEY);

CREATE TABLE items_tab OF
  item_t (itemnum PRIMARY KEY) ;

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

```

ALTER TABLE pedidos_tab
  ADD (SCOPE FOR (item) IS items_tab) ;

```

Esta última declaración sirve para restringir el dominio de los objetos referenciados desde `item` a aquellos que se almacenan en la tabla `items_tab`.

4.2.3 Inserción de objetos en las tablas

REM inserción en la tabla ITEMS_TAB*****

```

INSERT INTO items_tab VALUES (1004, 6750.00, 2);
INSERT INTO items_tab VALUES (1011, 4500.23, 2);
INSERT INTO items_tab VALUES (1534, 2234.00, 2);
INSERT INTO items_tab VALUES (1535, 3456.23, 2);
INSERT INTO items_tab VALUES (2004, 33750.00, 3);
INSERT INTO items_tab VALUES (3011, 43500.23, 4);
INSERT INTO items_tab VALUES (4534, 5034.00, 6);
INSERT INTO items_tab VALUES (5535, 34456.23, 5);

```

REM inserción en la tabla CLIENTES_TAB*****

Nótese como en estas definiciones se utilizan los constructores del tipo de objeto `direccion_t` y el tipo de colección `lista_tel_t`.

```

INSERT INTO clientes_tab
VALUES (
  1, 'Lola Caro',
  direccion_t('12 Calle Lisboa', 'Nules', 'CS',
    '12678'), lista_tel_t('415-555-1212')
) ;

INSERT INTO clientes_tab
VALUES (
  2, 'Jorge Luz',
  direccion_t('323 Calle Sol', 'Valencia', 'V',
    '08820'), lista_tel_t('609-555-1212','201-555-1212')
) ;

INSERT INTO clientes_tab
VALUES (
  3, 'Jose Perez',
  direccion_t('12 Calle Colon', 'Castellon', 'ES',
    '12001'), lista_tel_t('964-555-1212', '609-543-1212',
    '201-775-1212', '964-445-1212')
) ;

INSERT INTO clientes_tab
VALUES (
  4, 'Ana Gil',
  direccion_t('5 Calle Sueca', 'Burriana', 'ES',
    '12345'), lista_tel_t()
) ;

```

REM inserción en la tabla ORDENES_TAB*****

Nótese como en estas definiciones se utiliza el operador `REF` para obtener una referencia a

un objeto de `clientes_tab` y almacenarlo en la columna de otro objeto de `ordenes_tab`. La palabra clave `THE` se utiliza para designar la columna de las tuplas que cumplen la condición del `WHERE`, donde se deben realizar la inserción. Las tuplas que se insertan son las designadas por el segundo `SELECT`, y el objeto de la orden debe existir antes de comenzar a insertar líneas de pedido.

REM Ordenes del cliente 1*****

```
INSERT INTO ordenes_tab
  SELECT 1001, REF(C),
         SYSDATE, '10-MAY-1999',
         lineas_pedido_t(),
         NULL
  FROM   clientes_tab C
 WHERE  C.clinum= 1 ;
```

```
INSERT INTO THE (
  SELECT P.pedido FROM
    ordenes_tab P
  WHERE  P.ordnum = 1001
)
SELECT 01, REF(S), 12, 0
  FROM items_tab S WHERE
    S.itemnum = 1534;
```

```
INSERT INTO THE (
  SELECT P.pedido FROM
    ordenes_tab P
  WHERE  P.ordnum = 1001
)
SELECT 02, REF(S), 10, 10
  FROM items_tab S
 WHERE  S.itemnum = 1535;
```

REM Ordenes del cliente 2*****

```
INSERT INTO ordenes_tab
  SELECT 2001, REF(C),
         SYSDATE, '20-MAY-1999',
         lineas_pedido_t(),
         direccion_t('55 Madison
Ave', 'Madison', 'WI', '53715') FROM clientes_tab C
 WHERE  C.clinum= 2;
```

```
INSERT INTO THE (
  SELECT P.pedido FROM
    ordenes_tab P
  WHERE  P.ordnum = 2001
)
SELECT 10, REF(S), 1, 0
  FROM items_tab S WHERE
    S.itemnum = 1004;
```

```
INSERT INTO THE (
  SELECT P.pedido
  FROM   ordenes_tab P
```



```

        WHERE P.ordnum= 2001
    )
    VALUES( linea_t(11, NULL, 2, 1) ) ;

REM Ordenes del cliente 3*****

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

INSERT INTO THE (
SELECT P.pedido FROM
ordenes_tab P
WHERE P.ordnum = 3001
)
SELECT 30, REF(S), 18, 30
FROM   items_tab S
WHERE  S.itemnum = 3011;

INSERT INTO THE (
SELECT P.pedido FROM
ordenes_tab P
WHERE P.ordnum = 3001
)
SELECT 32, REF(S), 10, 100
FROM   items_tab S
WHERE  S.itemnum = 1535;

*****

INSERT INTO ordenes_tab
SELECT 3002, REF(C),
       SYSDATE, '15-JUN-1999',
       lineas_pedido_t(),
       NULL
FROM   clientes_tab C
WHERE  C.clinum= 3 ;

INSERT INTO THE (
SELECT P.pedido FROM
ordenes_tab P
WHERE P.ordnum = 3002
)
SELECT 34, REF(S), 200, 10
FROM   items_tab S
WHERE  S.itemnum = 4534;

REM Ordenes del cliente 4*****

INSERT INTO ordenes_tab
SELECT 4001, REF(C),
       SYSDATE, '12-MAY-1999',
       lineas_pedido_t(),
       direccion_t('34 Nave Oeste', 'Nules', 'CS', '12876')

```

```

FROM   clientes_tab C
WHERE  C.clinum= 4;

INSERT INTO THE (
  SELECT P.pedido FROM
    ordenes_tab P
  WHERE  P.ordnum = 4001
)
SELECT 41, REF(S), 10, 10
FROM items_tab S
WHERE  S.itemnum = 2004;

INSERT INTO THE (
  SELECT P.pedido FROM
    ordenes_tab P
  WHERE  P.ordnum = 4001
)
SELECT 42, REF(S), 32, 22
FROM items_tab S
WHERE  S.itemnum = 5535;

```

4.2.4 Borrado de los objetos, las tablas y los tipos de usuario

```

DELETE FROM ordenes_tab;
DROP TABLE ordenes_tab;

DELETE FROM clientes_tab;
DROP TABLE clientes_tab;

DELETE FROM items_tab;
DROP TABLE items_tab;

DROP TYPE ordenes_t;
DROP TYPE lineas_pedido_t;
DROP TYPE linea_t;
DROP TYPE item_t;

DROP TYPE clientes_t;
DROP TYPE lista_tel_t;
DROP TYPE direccion_t;

```

4.2.5 Definición de métodos para los tipos

El siguiente método calcula la suma de los valores de las líneas de pedido de la orden de pedido sobre la que se ejecuta.

```

CREATE TYPE ordenes_t AS OBJECT (
  ordnum          NUMBER,
  cliente         REF clientes_t,
  fechpedido     DATE,
  fechentrega    DATE,
  pedido         lineas_pedido_t,
  direcentrega   direccion_t,

  MEMBER FUNCTION
    coste_total RETURN NUMBER,
  PRAGMA RESTRICT_REFERENCES(coste_total, WNDS, WNPS) );

```

```

CREATE TYPE BODY ordenes_t AS
  MEMBER FUNCTION coste_total RETURN NUMBER IS
    i      INTEGER;
    item   item_t;
    linea  linea_t;
    total  NUMBER:=0;

  BEGIN
    FOR i IN 1..SELF.pedido.COUNT LOOP

      linea:=SELF.pedido(i);
      SELECT Deref(linea.item) INTO item FROM DUAL;
      total:=total + linea.cantidad * item.precio;
    END LOOP;
    RETURN total;
  END;
END;

```

- La palabra clave **SELF** permite referirse al objeto sobre el que se ejecuta el método.
- La palabra clave **COUNT** sirve para contar el número de elementos de una tabla o de un array. Junto con la instrucción **LOOP** permite iterar sobre los elementos de una colección, en nuestro caso las líneas de pedido de una orden.
- El **SELECT** es necesario porque Oracle no permite utilizar **Deref** directamente en el código PL/SQL.

4.2.6 Consultas a la base de datos anterior

1- Consultar la definición de la tabla de clientes.

```
SQL> describe clientes_tab;
```

Name	Null?	Type
CLINUM	NOT NULL	NUMBER
CLINOMB		VARCHAR2(200)
DIRECCION		DIRECCION_T
LISTA_TEL		LISTA_TEL_T

2- Insertar en la tabla de clientes a un nuevo cliente con todos sus datos.

```

SQL> insert into clientes_tab
  2 values(5, 'John smith',
  3 direccion_t('67 rue de percebe', 'Gijon', 'AS', '73477'),
  4 lista_tel_t('7477921749', '83797597827'));

```

```
1 row created.
```

3- Consultar y modificar el nombre del cliente número 2.

```
SQL> select clinomb from clientes_tab where clinum=2;
```

```
CLINOMB
```

```
-----
Jorge Luz
```



```
SQL> update clientes_tab
  2 set clinomb='Pepe Puig' where clinum=5;

1 row updated.
```

4- Consultar y modificar la dirección del cliente número 2.

```
SQL> select direccion from clientes_tab where clinum=2;
```

```
DIRECCION(CALLE, CIUDAD, PROV, CODPOS)
```

```
-----
DIRECCION_T('Calle Sol', 'Valencia', 'VA', '08820')
```

```
SQL> update clientes_tab
  2 set direccion=direccion_t('Calle Luna', 'Castello', 'CS', 68734')
  3 where clinum=2;
```

```
1 row updated.
```

5- Consultar todos los datos del cliente número 1 y añadir un nuevo teléfono a su lista de teléfonos.

```
SQL> select * from clientes_tab where clinum=1;
```

```
CLINUM
```

```
-----
CLINOMB
```

```
-----
DIRECCION(CALLE, CIUDAD, PROV, CODPOS)
```

```
-----
LISTA_TEL
```

```
-----
1
Lola Caro
DIRECCION_T('Calle Luna', 'Castellon', 'CS',
'64827') LISTA_TEL_T('415-555-1212')
```

También se puede consultar así:

```
SQL> select value(C) from clientes_tab C where C.clinum=1;
```

```
VALUE(C) (CLINUM, CLINOMB, DIRECCION(CALLE, CIUDAD, PROV,
CODPOS), LISTA_TEL)
```

```
-----
CLIENTES_T(1, 'Lola Caro', DIRECCION_T('Calle Luna', 'Castellon',
'CS', '64827'), LISTA_TEL_T('415-555-1212'))
```

```
SQL> update clientes_tab
  2 set lista_tel=lista_tel_t('415-555-1212', '6348635872')
  3 where clinum=1;

1 row updated.
```

6- Visualizar el nombre del cliente que ha realizado la orden número 1001.

```
SQL> select o.cliente.clinomb from ordenes_tab o where o.ordnum=1001;
```

```
CLIENTE.CLINOMB
```

```
-----
Lola Caro
```

7- Visualizar todos los detalles del cliente que ha realizado la orden número 1001.

```
SQL> select Deref(o.cliente) from ordenes_tab o where o.ordnum=1001;

Deref(O.CLIENTE) (CLINUM, CLINOMB, DIRECCION(CALLE, CIUDAD, PROV, CODPOS),
LISTA_TEL
-----
CLIENTES_T(1, 'Lola Caro', DIRECCION_T('Calle Luna', 'Castellon',
'CS', '64827'), LISTA_TEL_T('415-555-1212', '6348635872'))
```

De la siguiente manera se obtiene la referencia al objeto, la cuál es ininteligible.

```
SQL> select o.cliente from ordenes_tab o where o.ordnum=1001;

CLIENTE
-----
00002EA5F6693E4A73F8E003960286473F83EA5F6693E3E73F8E0039680286473F8
```

8- Visualizar el número de todos los items que se han pedido en la orden número 3001.

```
SQL> select cursor(select p.item.itemnum from table(o.pedido)
p) 2 from ordenes_tab o where o.ordnum=3001;

CURSOR(SELECTP.ITEM.
-----
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

ITEM.ITEMNUM
-----
3011
1535
```

9- Seleccionar el número de orden y el coste total de las ordenes hechas por el cliente número 3.

```
SQL> select o.ordnum, o.coste_total() from ordenes_tab
o 2 where o.cliente.clinum=3;

ORDNUM O.COSTE_TOTAL()
-----
3001      817566.44
3002      1006800
```