



VNIVERSITATIS VALÈNCIA

Departament d'Informàtica

**BASES DE DATOS
OBJETO-RELACIONALES**

Un sistema de gestión de pedidos

Wladimiro Díaz Villanueva

Curso 2001-2002

Índice General

1	Introducción	3
1.1	Entidades y relaciones	3
2	Aproximación relacional	4
2.1	Tablas	4
2.2	Inserción de valores	5
2.3	Selección de valores	6
2.4	Modificación de valores	6
2.5	Borrado	7
3	Modelo objeto-relacional	7
3.1	La vía objeto-relacional	7
3.2	Definición de tipos	8
3.2.1	Declaración del tipo <code>item_t</code>	9
3.2.2	Declaración del tipo <code>pedido_t</code>	9
3.2.3	Declaración del tipo <code>stock_t</code>	10
3.3	Definición de métodos	10
3.4	Creación de las tablas de objetos	12
3.4.1	La tabla de objetos <code>clientes_tab</code>	12
3.4.2	La tabla de objetos <code>stock_tab</code>	12
3.4.3	La tabla de objetos <code>pedidos_tab</code>	12
3.5	Modificación de las tablas	13
3.6	Inserción de valores	13
3.6.1	<code>stock_tab</code>	14
3.6.2	<code>clientes_tab</code>	14
3.6.3	<code>pedidos_tab</code>	14
3.7	Selección de valores	16
3.8	Borrado de valores	16

Un sistema de gestión de pedidos

1 Introducción

Este ejemplo está basado en una actividad empresarial simple: gestión de pedidos de usuarios. El ejemplo se estructura en dos partes. En la primera parte se implementa el esquema utilizando sólo los tipos de datos intrínsecos de Oracle. Denominaremos a esta parte como *aproximación relacional*. Mediante esta aproximación, se crean tablas que contienen los datos de la aplicación y se utilizan técnicas bien conocidas para implementar las relaciones entre entidades.

En la segunda parte se utilizan tipos de datos definidos por el usuario para trasladar las entidades y las relaciones directamente en esquemas de objetos que pueden ser manipulados por el DBMS. Denominaremos a esta técnica la *aproximación objeto-relacional*.

1.1 Entidades y relaciones

Las entidades básicas de este ejemplo son:

- Los clientes (compradores de productos).
- El *stock* de productos a la venta.
- Los pedidos de compra.

Podemos identificar las siguientes relaciones en el modelo:

- Los clientes tienen una relación uno-a-muchos con los pedidos de compra ya que un comprador puede solicitar varios pedidos, pero un pedido determinado está asociado a un sólo cliente.
- Los pedidos mantienen una relación muchos-a-muchos con los items del *stock* ya que un pedido puede contener varios items del *stock* y un item del *stock* puede aparecer en múltiples pedidos.

La forma usual de abordar la relación muchos-a-muchos entre pedidos y el *stock* es introducir otra entidad denominada *lista de items*. Un pedido puede tener un número arbitrario de items, pero cada línea de la lista de *items* pertenece a un único pedido. Un elemento del *stock* puede aparecer en un gran número de listas de items, pero cada línea de la lista de items hace referencia sólo a un item del *stock*.

En la tabla 1 se muestra la información requerida para cada una de las entidades descritas.

El problema es que los atributos de las entidades en el mundo real son complejos y por lo tanto, requieren un conjunto complejo de atributos para contener las estructuras de datos. Por ejemplo:

Entidad	Información requerida
Cliente	Información de contacto
Stock	Identificador del item, precio, importe del IVA
Pedido	Cliente, pedido y fecha de envío, dirección de envío
Lista de items	Item del <i>stock</i> , cantidad, precio, descuento aplicable para cada item

Tabla 1: Información requerida para cada una de las entidades del modelo de gestión de pedidos de compra

- Una dirección contiene atributos tales como la calle, la población, la provincia y el código postal.
- Un cliente puede tener varios números de teléfono.
- La lista de items contiene atributos y simultáneamente es un atributo del pedido.

Los tipos predefinidos estándar no pueden representar estos elementos directamente. La aproximación objeto-relacional hace posible manejar estas estructuras complejas de forma más eficiente.

2 Aproximación relacional

El modelo relacional normaliza las entidades y sus atributos y estructura las entidades comprador, pedidos de compra y *stock* en tablas. Las direcciones se parten en componentes estándar. Se introduce un número arbitrario de números de teléfono que un cliente puede tener y le asigna una columna a cada uno de ellos.

El modelo relacional separa las listas de items de sus pedidos y las pone en su propia tabla individual. La tabla contiene columnas para las claves externas a las tablas *stock* y pedido.

2.1 Tablas

```
CREATE TABLE clientes (
  numclient NUMBER,
  nombre    VARCHAR2(200),
  calle     VARCHAR2(200),
  poblacion VARCHAR2(200),
  provincia VARCHAR2(50),
  cod_zip   VARCHAR2(20),
  telef1    VARCHAR2(20),
  telef2    VARCHAR2(20),
  telef3    VARCHAR2(20),
  PRIMARY KEY (numclient)
);

CREATE TABLE pedidos (
  numpedido NUMBER,
  numclient  NUMBER REFERENCES clientes,
  fecha_pedido DATE,
```

```

    fecha_envio DATE,
    calle_envio VARCHAR2(200),
    pobla_envio VARCHAR2(200),
    prov_envio VARCHAR2(50),
    zip_envio VARCHAR2(20),
    PRIMARY KEY (numpedido)
);

CREATE TABLE stock (
    numstock NUMBER PRIMARY KEY,
    precio NUMBER,
    cod_tasa NUMBER
);

CREATE TABLE lista_items (
    numitem NUMBER,
    numpedido NUMBER REFERENCES pedidos,
    numstock NUMBER REFERENCES stock,
    cantidad NUMBER,
    descuento NUMBER,
    PRIMARY KEY (numpedido, numitem)
);

```

La primera tabla, `clientes`, contiene información acerca de los compradores. No hace referencia a otras tablas, pero la tabla `pedidos` contiene una columna `numclient`, que contiene una clave externa a la tabla `pedidos`. La clave externa implementa la relación muchos-a-uno entre `pedidos` y `clientes`.

La tabla `lista_items` contiene las claves externas `numpedido` y `numstock` que hacen referencia a las tablas `pedido` y `stock` respectivamente.

2.2 Inserción de valores

En una aplicación basada en las tablas definidas en la sección anterior, sentencias como las que siguen insertan valores en las tablas:

```

INSERT INTO clientes
VALUES (1, 'Juan Perez', 'Avda. Camelias, 12-7',
        'Valencia', 'Valencia', '46018',
        '96 123 1212', NULL, NULL);

INSERT INTO clientes
VALUES (2, 'Isabel Arias', 'C/ Colegio Mayor, 5-1',
        'Burjassot', 'Valencia', '46100',
        '96 354 3232', '96 354 3233', NULL);

INSERT INTO pedidos
VALUES (1001, 1, SYSDATE, '15-MAY-2001',
        NULL, NULL, NULL, NULL);

INSERT INTO pedidos
VALUES (2001, 2, SYSDATE, '20-MAY-2001',
        'Avda. Americas, 3-3', 'Valencia',
        'Valencia', '46008');

```

```

INSERT INTO stock VALUES(1004, 6750.0, 2);
INSERT INTO stock VALUES(1011, 4500.0, 2);
INSERT INTO stock VALUES(1534, 2235.0, 2);
INSERT INTO stock VALUES(1535, 3455.0, 2);

INSERT INTO lista_items VALUES(01, 1001, 1534, 12, 0);
INSERT INTO lista_items VALUES(02, 1001, 1535, 10, 10);
INSERT INTO lista_items VALUES(10, 2001, 1004, 1, 0);
INSERT INTO lista_items VALUES(11, 2001, 1011, 2, 1);

```

2.3 Selección de valores

Suponiendo que los valores se han insertado en las tablas de la forma usual, las aplicaciones pueden ahora ejecutar consultas de diverso tipo para recuperar la información de los datos almacenados. Algunos ejemplos son:

- Cliente y datos del pedido para la orden de compra 1001:

```

SELECT C.numclient, C.nombre, C.calle, C.poblacion,
       C.provincia, C.cod_zip, C.telef1, C.telef2,
       C.telef3,
       P.numpedido, P.fecha_pedido,
       L.numstock, L.numitem, L.cantidad, L.descuento
FROM clientes C, pedidos P, lista_items L
WHERE C.numclient = P.numclient
      AND P.numpedido = L.numpedido
      AND P.numpedido = 1001;

```
- Valor total de cada pedido de compra:

```

SELECT P.numpedido, SUM(S.precio * L.cantidad)
FROM pedidos P, items L, stock S
WHERE P.numpedido = L.numpedido
      AND L.numstock = S.numstock
GROUP BY P.numpedido;

```
- Pedidos de compra e información de la lista de items del elemento del stock 1004:

```

SELECT P.numpedido, P.numclient,
       L.numstock, L.numitem, L.cantidad, L.descuento
FROM pedidos P, lista_items L
WHERE P.numpedido = L.numpedido
      AND L.numstock = 1004;

```

2.4 Modificación de valores

Siguiendo con el modelo relacional descrito, para modificar la cantidad de items del stock 1001 para el pedido 01 se realizaría del siguiente modo:

```

UPDATE lista_items
SET cantidad = 20
WHERE numpedido = 1
AND numstock = 1001;

```

2.5 Borrado

Para eliminar el pedido 1001 utilizaríamos la siguiente sentencia SQL:

```
DELETE
  FROM lista_items ,
  WHERE numpedido = 1001;
```

```
DELETE
  FROM pedidos
  WHERE numpedido = 1001;
```

3 Modelo objeto-relacional

Las aplicaciones escritas en lenguajes de tercera generación, como C++, son capaces de implementar tipos definidos por el usuario de gran complejidad, encapsulando los datos y los métodos. Por contra, el SQL sólo proporciona tipos básicos escalares y ningún método para encapsular las operaciones relevantes.

De acuerdo con esto, se nos plantea la siguiente pregunta: ¿Por qué no desarrollar las aplicaciones en lenguajes de tercera generación? Fundamentalmente por dos motivos:

- Los sistemas de gestión de bases de datos proporcionan funcionalidades que para replicar se requerirían millones de horas de trabajo.
- Uno de los problemas a la hora de desarrollar aplicaciones de gestión utilizando un lenguaje 3GL es que estos carecen de persistencia.

Esto deja al desarrollador de aplicaciones con el problema de simular tipos complejos empleando sólo la implementación incompleta que permite el SQL. Esto introduce, aparte de las horas de trabajo que implica, serios problemas de implementación, ya que obliga a:

- Traducir de la lógica de la aplicación a la lógica del sistema de almacenamiento a la hora de escribir los datos.
- Llevar a cabo el proceso inverso cuando los datos se requieren de nuevo en una aplicación.

Por supuesto, se produce un intenso tráfico bidireccional entre el espacio de direcciones de la aplicación y el del servidor, con el consiguiente decremento del rendimiento. El objetivo de la tecnología objeto-relacional es resolver estos problemas.

3.1 La vía objeto-relacional

La aproximación objeto-relacional del ejemplo que hemos considerado comienza con las mismas entidades y relaciones esbozadas en el apartado 1.1, pero la utilización de tipos definidos por el usuario permite trasladar más información de estas estructuras al esquema de la base de datos.

Algunas de las mejoras que podemos introducir mediante el modelo objeto-relacional son:

- En lugar de separar las direcciones o los números de teléfono de contacto en columnas desligadas en tablas relacionales, el modelo objeto-relacional define tipos para representar estas entidades.
- Del mismo modo, en lugar de separar las listas de ítems en tablas separadas, el modelo objeto relacional permite mantener la listas con sus respectivos pedidos como tablas anidadas.
- En el modelo objeto-relacional, las entidades principales — `clientes`, `stock` y `pedidos` — se convierten en objetos. Las relaciones muchos-a-uno se representan mediante referencias entre estos objetos.

3.2 Definición de tipos

Las siguientes sentencias sientan las bases:

```
CREATE TYPE item_t;  
CREATE TYPE pedido_t;  
CREATE TYPE stock_t;
```

Las sentencias anteriores definen tipos de datos incompletos. Esto le indica a Oracle que las definiciones completas se darán posteriormente.

La siguiente sentencia define un tipo vector, `lista_tel_t`, cuyos elementos son hasta 10 números de teléfono almacenados en un `VARCHAR2`. En este caso, la lista representa un conjunto de números de teléfono de contacto de un único cliente:

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

Una lista de números de teléfono podría definirse como un vector o como una tabla anidada. En este caso, un vector `VARRAY` es mejor elección que una tabla anidada por los siguientes motivos:

- El orden de los números puede ser importante. Un `VARRAY` está ordenado mientras que las tablas anidadas no.
- El número de números de teléfono para un cliente concreto es pequeño. Los vectores `VARRAY` obligan a especificar el número de elementos con antelación y por tanto gestionan el almacenamiento de forma mucho más eficiente (ya que no tienen ninguna limitación especial en el tamaño).
- No existe ningún motivo especial por el cual estemos interesados en “preguntar” por la lista de números de teléfono, por lo que el uso de una tabla anidada no ofrece ningún beneficio adicional.

En general, si el orden y los límites no son consideraciones importantes en el diseño, se debe usar la siguiente receta para decidir entre `VARRAYs` y tablas anidadas:

- Si se requiere consultar la colección, es mejor utilizar tablas anidadas.
- Si sólo se requiere recuperar la colección como un todo, es aconsejable utilizar vectores `VARRAY`.

La siguiente sentencia define el tipo de objeto `direccion_t` que se utilizará para representar direcciones postales. Todos los atributos son de tipo cadena de caracteres y representan las componentes comunes de una dirección de correo ligeramente simplificada:

```
CREATE TYPE direccion_t AS OBJECT (  
    calle      VARCHAR2(100),  
    poblacion VARCHAR2(100),  
    provincia  VARCHAR2(50),  
    cod_zip    VARCHAR2(20)  
);
```

La siguiente sentencia define un tipo de objeto que utiliza los tipos ya definidos como bloques estructurales. El objeto también implementa un método de comparación:

```
CREATE TYPE cliente_t AS OBJECT (  
    numclient  NUMBER,  
    nombre     VARCHAR2(200),  
    direccion  direccion_t,  
    telefonos  lista_tel_t,  
    ORDER MEMBER FUNCTION  
        ord_cliente(x IN cliente_t) RETURN INTEGER,  
    PRAGMA RESTRICT_REFERENCES (  
        ord_cliente, WNDS, WNPS, RNPS, RNDS)  
);
```


Las entidades del tipo `cliente_t` definido son objetos que representan bloques de información acerca de un cliente concreto. Los atributos de este tipo son un número, una cadena de caracteres, un objeto de tipo `direccion_t` y un `VARRAY` de tipo `lista_tel_t`.

Cada objeto `cliente_t` tiene también asociado un método de clasificación, uno de los dos tipos de métodos de comparación. Siempre que Oracle requiera comparar dos objetos de tipo `cliente_t` invocará al método `ord_cliente`. La sentencia anterior no incluye el programa PL/SQL que implementa el método. Este se definirá con posterioridad.

3.2.1 Declaración del tipo `item_t`

La siguiente sentencia completa la definición del tipo de objeto incompleto `item_t` definido al principio de esta sección:

```
CREATE TYPE item_t AS OBJECT (  
    numitem NUMBER,  
    stockref REF stock_t,  
    cantidad NUMBER,  
    descuento NUMBER  
);
```

La siguiente sentencia define un tipo de tabla denominado `lista_item_t`. Un dato de este tipo es una tabla anidada en la que cada fila contiene un objeto `item_t`.

```
CREATE TYPE lista_item_t AS TABLE OF item_t;
```

Para representar un valor múltiple como la lista de items de un pedido, una tabla anidada es mejor elección que un `VARRAY` de objetos `item_t` debido a los siguientes motivos:

- Una de las operaciones que llevarán a cabo muchas aplicaciones será consultar el contenido de una lista de items. Esta operación es poco eficiente en `VARRAYS` ya que implica convertir esta estructura en una tabla anidada.
- Cabe esperar que algunas aplicaciones requieran indexar los datos de la lista de items. Esto es posible en tablas anidadas pero no en `VARRAYS`.
- El orden de los items en la lista de items carece de importancia y el número de item puede utilizarse para ordenar los items en caso de que sea necesario.
- No hay un límite práctico en el número de items en un pedido. Utilizar un `VARRAY` obliga a especificar un valor máximo en el número de elementos.

3.2.2 Declaración del tipo `pedido_t`

La siguiente sentencia completa la definición del tipo incompleto `pedido_t` declarado al principio de esta sección:

```
CREATE TYPE pedido_t AS OBJECT (  
    numpedido NUMBER,  
    clienteref REF cliente_t,  
    fechapedido DATE,  
    fechaenvio DATE,  
    lista_item lista_item_t,  
    direc_envio direccion_t,  
    MAP MEMBER FUNCTION  
        valor RETURN NUMBER,  
    PRAGMA RESTRICT_REFERENCES (  
        valor, WNDS, WNPS, RNPS, RNDS),  
    MEMBER FUNCTION
```

```

    valor_total RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (
        valor_total, WNDS, WNPS)
);

```

Las instancias de este tipo son objetos que representan un pedido. Contienen seis atributos, incluyendo una referencia (REF), una tabla anidada de tipo `lista_item_t` y un objeto de tipo `direccion_t`.

También cuentan con dos métodos: `valor` y `valor_total`. El primero es un método de MAP (proyección), una de las dos clases de clasificación. Un método de MAP devuelve la posición relativa del objeto, de modo que cada vez que Oracle necesita comparar dos objetos de tipo `pedido_t` invoca implícitamente a este método. Las declaraciones PRAGMA proporcionan información al PL/SQL acerca del tipo de acceso a la base de datos que requieren.

3.2.3 Declaración del tipo `stock_t`

Por último, la siguiente sentencia completa la definición del tercer tipo, `stock_t`, definido al principio de la sección:

```

CREATE TYPE stock_t AS OBJECT (
    numstock NUMBER,
    precio NUMBER,
    cod_tasa NUMBER
);

```

Las instancias del tipo `stock_t` representan los elementos del *stock* que los clientes solicitan en sus pedidos.

3.3 Definición de métodos

En este apartado mostraremos cómo se especifican los métodos de los tipos de objeto `cliente_t` y `pedido_t`, es decir, el programa PL/SQL que implementa los métodos:

```

CREATE OR REPLACE TYPE BODY pedido_t AS
    MEMBER FUNCTION valor_total RETURN NUMBER IS
        i          INTEGER;
        stock      stock_t;
        item       item_t;
        total      NUMBER := 0;
        precio     NUMBER;
    BEGIN
        FOR i IN 1..SELF.lista_item.COUNT LOOP
            item := SELF.lista_item(i);
            SELECT Deref(item.stockref) INTO stock FROM DUAL;
            total := total + item.cantidad * stock.precio;
        END LOOP;
        RETURN total;
    END;
    MAP MEMBER FUNCTION valor RETURN NUMBER IS
    BEGIN
        RETURN numpedido;
    END;
END;

```

- El método `valor` es sencillo: simplemente devuelve el número asociado a cada pedido.

- El método `valor_total` usa una serie de mecanismos objeto-relacionales para devolver la suma de los precios de los elementos de la lista de items asociada al pedido.
 - Como ya hemos dicho, la misión de esta función es calcular el importe total de los elementos asociados al pedido. La palabra clave `SELF`, que es un parámetro implícito en cada función, permite hacer referencia al objeto.
 - La palabra clave `COUNT` proporciona el número total de elementos en un vector o tabla PL/SQL. En combinación con `LOOP` permite a la aplicación iterar sobre todos los elementos de la colección; en este caso los items del pedido. De este modo, `SELF.lista_item.COUNT` cuenta el número de elementos en la tabla anidada que coincide con el atributo `lista_item` del objeto `pedido_t` representado por `SELF`.
 - El operador `DEREF` toma un valor de tipo referencia como argumento y devuelve un objeto fila. En este caso, `DEREF(line_item.stockref)` toma el atributo `stockref` como argumento y devuelve un objeto `stock_t`. Volviendo a nuestra definición de los datos, podemos comprobar que `stockref` es un atributo del objeto `item_t` que es a su vez un atributo de `lista_item_t`. Este objeto-lista, que hemos estructurado como una tabla anidada es al fin y al cabo un atributo del objeto `pedido_t` representado por `SELF`. Esto puede parecer realmente complejo, pero basta retomar nuestra perspectiva del mundo real para clarificar la situación: un pedido (`pedido_t`) está constituido por una lista (`lista_item_t`) de items (`item_t`), cada uno de los cuales contiene una referencia (`stockref`) a información acerca del item (`stock_t`). La operación que hemos llevado a cabo simplemente recupera la información vía un mecanismo objeto-relacional.
 - Todas estas entidades son tipos de datos abstractos y como tales pueden interpretarse como patrones de objetos. ¿Cómo recuperamos entonces los valores actuales de los objetos `stock`? Se requiere, por tanto, la sentencia SQL `SELECT` con la llamada explícita a `DEREF` ya que Oracle no permite la derreferencia implícita de `REFs` en programas PL/SQL. La variable PL/SQL `stock` es del tipo `stock_t`. La sentencia `SELECT` le asigna el valor representado por `DEREF(line_item.stockref)` y ese objeto es el item del stock referenciado en el *i*-ésimo elemento de la lista de items.
 - Habiendo recuperado el item del stock en cuestión, el siguiente paso es calcular el importe. El programa hace referencia al coste del item como `stock.precio`. Sin embargo, para calcular el precio del item también se requiere conocer la cantidad de elementos pedidos. En nuestra aplicación, el término `line_item.cantidad` representa al atributo `cantidad` de cada objeto `item_t`.

El resto del método es fácil de comprender. El bucle suma los valores de los elementos de la lista y finalmente devuelve el importe total.

El método `ord_cliente` del tipo de objeto `cliente_t` se define a continuación:

```
CREATE OR REPLACE TYPE BODY cliente_t AS
ORDER MEMBER FUNCTION
  ord_cliente (x IN cliente_t) RETURN INTEGER IS
BEGIN
  RETURN numclient - x.numclient;
END;
END;
```

Como ya hemos mencionado, la función `ord_cliente` sirve para comparar dos pedidos de clientes. El mecanismo de operación es sencillo. El método utiliza otro objeto de tipo `cliente_t` como argumento de entrada y devuelve la diferencia entre los números `numclient`. Si el número de cliente tiene algún significado lógico (por ejemplo, números menores significan clientes más antiguos), el valor devuelto por esta función permite ordenar los objetos de este tipo. Por tanto, el método puede devolver:

- Un número negativo si cliente actual es más antiguo que `x`.

- Un número positivo si el cliente actual es más reciente.
- Cero, en cuyo caso el objeto está comparándose consigo mismo.

3.4 Creación de las tablas de objetos

Generalmente, podemos considerar que la relación entre “objetos” y “tablas” es la siguiente:

- Las clases, que representan a las entidades, cumplen la misma función que las tablas.
- Los atributos son equivalentes a las columnas.
- Los objetos son equivalentes a las filas.

Contemplado desde esta perspectiva, cada tabla se puede considerar un tipo implícito cada uno de cuyos objetos (las filas específicas) comparte los mismos atributos (las columnas). De acuerdo con esto, la creación de tipos de datos abstracto de forma explícita introduce un nuevo nivel de funcionalidad.

3.4.1 La tabla de objetos `clientes_tab`

La siguiente sentencia define la tabla de objetos `clientes_tab` que contiene objetos del tipo `cliente_t`:

```
CREATE TABLE clientes_tab OF cliente_t
  (numclient PRIMARY KEY);
```

Es importante destacar que existe una diferencia sintáctica en la definición de las tablas de objetos: se utiliza el término `OF`.

Dos son los aspectos relevantes que proporciona la tecnología objeto relacional y que vamos a comentar:

- **Los tipos de dato se comportan como patrones para las tablas de objetos:** El hecho de que exista un tipo `cliente_t` significa que es posible crear numerosas tablas de este tipo. Por ejemplo, podríamos crear una tabla `clientes_2_tab`. Sin esta capacidad, sería necesario definir cada tabla de forma individual.

Por otra parte, la capacidad para crear tablas del mismo tipo no implica necesariamente que no se puedan introducir variaciones. Por ejemplo, en la sentencia que crea la tabla `clientes_tab` se define como clave primaria la columna `numclient`. Cualquier otra tabla de objetos `cliente_t` no tiene porque satisfacer esta restricción. Esto pone de manifiesto un punto importante: las restricciones se aplican a las tablas, no a la definición de tipos.

- **Las tablas de objetos pueden contener objetos embebidos:** Examinando la definición de la tabla `clientes_tab` podemos observar que la columna `direccion` contiene objetos del tipo `direccion_t`. En otras palabras: un tipo de datos abstracto puede contener atributos que son a su vez tipos de dato abstractos. Cuando un tipo de dato es instanciado como un objeto, los objetos incluidos también son instanciados. En la tabla de objetos `clientes_tab`, Oracle crea una columna para los objetos `direccion_t` y sus atributos. Es posible hacer referencia a estos elementos utilizando la notación-punto, por ejemplo, para construir un índice en la columna `cod_zip` podemos hacer referencia a él mediante `direccion.cod_zip`.

3.4.2 La tabla de objetos `stock_tab`

La siguiente sentencia crea la tabla de objetos `stock_tab`:

```
CREATE TABLE stock_tab OF stock_t
  (numstock PRIMARY KEY);
```

Esto no introduce nada nuevo: esta sentencia simplemente crea una tabla cuyas columnas son objetos del tipo `stock_t`.

3.4.3 La tabla de objetos pedidos_tab

La siguiente sentencia define una tabla constituida por objetos del tipo pedido_t:

```
CREATE TABLE pedidos_tab OF pedido_t (  
    PRIMARY KEY ( numpedido ),  
    SCOPE FOR ( clientref ) IS clientes_tab  
)  
NESTED TABLE lista_item STORE AS lista_item_tab ;
```

Cada fila de la tabla es un objeto de tipo pedido_t, cuyos atributos son:

numpedido	NUMBER
clientref	REF cliente_t
fechapedido	DATE
fechaenvio	DATE
lista_item	lista_item_t

En este punto hemos introducido dos nuevos elementos:

- El “scope” asociado al operador REF de la columna clientref. Cuando no se aplican restricciones en el ámbito, el operador REF permite referenciar a cualquier objeto-fila. Sin embargo, las referencias clientref pueden hacer referencia sólo a filas en la tabla clientes_tab. Esta limitación sólo afecta a las columnas clientref de la tabla pedidos_tab, pero no es aplicable a los atributos de todos los objetos pedido_t que puedan almacenarse en cualquier otra tabla.
- El segundo elemento introducido tiene que ver con el hecho de que cada fila cuenta con una columna que es una tabla anidada lista_item_t. La última sentencia crea una tabla lista_item_tab para contener las columnas lista_item_t de todas las filas de la tabla pedidos_tab.

Todas las filas de una tabla anidada se encuentran en una tabla de almacenamiento separada. Se utiliza una fila oculta en esta tabla de almacenamiento, denominada NESTED_TABLE_ID, para relacionar las filas con su correspondiente fila padre, de modo que todos los elementos de una tabla anidada que pertenecen a un determinado padre tienen el mismo valor del índice NESTED_TABLE_ID.

3.5 Modificación de las tablas

La siguiente sentencia modifica la tabla de almacenamiento lista_item_tab, que contiene las columnas lista_item_t de la tabla de objetos anidada en pedidos_tab, para imponer una restricción en el ámbito de las referencias (REFs) que contiene:

```
ALTER TABLE lista_item_tab  
    ADD ( SCOPE FOR ( stockref ) IS stock_tab );
```

La tabla de almacenamiento lista_item_tab contiene columnas anidadas del tipo lista_item_t. Uno de los atributos de este objeto, y por tanto de cada columna de la tabla lista_item_tab, es stockref que es de tipo REF stock_t. La sentencia ALTER anterior restringe el ámbito de la columna de referencias a la tabla de objetos stock_tab.

La siguiente sentencia modifica de nuevo la tabla anidada lista_item_tab para especificar un índice:

```
ALTER TABLE lista_item_tab  
    STORAGE ( NEXT 5K PCTINCREASE 5 MINEXTENTS 1 MAXEXTENTS 20 );
```

```
CREATE INDEX lista_item_ind  
    ON lista_item_tab ( numitem, NESTED_TABLE_ID );
```

Como se ha comentado antes, la tabla de almacenamiento de una tabla anidada en una columna de una tabla de objetos contiene una columna oculta denominada NESTED_TABLE_ID. La sentencia anterior crea un índice en esa columna, haciendo más eficiente el acceso a los contenidos de la columnas de tipo lista_item_tab.

3.6 Inserción de valores

Las siguientes líneas de código muestran como insertar la información en las tablas de objetos creadas.

3.6.1 stock_tab

```
INSERT INTO stock_tab VALUES (1004, 6750.0, 2);
INSERT INTO stock_tab VALUES (1011, 4500.0, 2);
INSERT INTO stock_tab VALUES (1534, 2235.0, 2);
INSERT INTO stock_tab VALUES (1535, 3455.0, 2);
```

3.6.2 clientes_tab

```
INSERT INTO clientes_tab
VALUES (
    1, 'Juan Perez',
    direccion_t('Avda. Camelias, 12-7', 'Valencia',
    'Valencia', '46018'),
    lista_tel_t('96 123 1212')
);

INSERT INTO clientes_tab
VALUES (
    2, 'Isabel Arias',
    direccion_t('C/ Colegio Mayor, 5-1', 'Burjassot',
    'Valencia', '46100'),
    lista_tel_t('96 354 3232', '96 354 3233')
);
```

3.6.3 pedidos_tab

```
INSERT INTO pedidos_tab
SELECT 1001, REF(c),
    SYSDATE, '15-MAY-2001',
    lista_item_t(),
    NULL
FROM clientes_tab c
WHERE c.numclient = 1 ;
```

La sentencia anterior construye un objeto de tipo `pedido_t` con los siguientes atributos:

<code>numpedido</code>	1001
<code>clientref</code>	REF al cliente número 1
<code>fechapedido</code>	SYSDATE
<code>fechaenvio</code>	15 de mayo de 2001
<code>lista_item</code>	Una lista de items vacía
<code>direc_envio</code>	NULL

En esta sentencia se ha utilizado una consulta para construir una referencia a la fila-objeto en la tabla `clientes_tab` cuyo valor de `numclient` es igual a 1.

La siguiente sentencia utiliza una subconsulta, marcada con la palabra clave `THE`, para identificar el objetivo de la inserción, formalmente la tabla anidada en la columna `lista_item` de la fila de la tabla `pedidos_tab` cuyo valor de `numpedido` es 1001:

```
INSERT INTO THE (
    SELECT p.lista_item
    FROM pedidos_tab p
```

```

        WHERE p.numpedido = 1001
    )
SELECT 01, REF(s), 12, 0
FROM stock_tab s
WHERE s.numstock = 1534;

```

La palabra clave THE se utiliza para manipular las filas individuales de una tabla anidada almacenada en una columna. THE es el prefijo de una subconsulta que devuelve una columna simple o una tabla anidada. La necesidad de utilizar la palabra clave THE obedece al hecho de que es necesario informar a Oracle de que el resultado de la consulta no es un valor escalar.

El resto de los valores se insertan de forma muy similar:

```

INSERT INTO pedidos_tab
SELECT 2001, REF(c),
      SYSDATE, '20-MAY-2001',
      lista_item_t(),
      direccion_t('Avda. Americas, 3-3', 'Valencia',
                 'Valencia', '46008')
FROM clientes_tab c
WHERE c.numclient = 2;

```

```

INSERT INTO THE (
SELECT p.lista_item
FROM pedidos_tab p
WHERE p.numpedido = 1001
)
SELECT 02, REF(s), 10, 10
FROM stock_tab s
WHERE s.numstock = 1535;

```

```

INSERT INTO THE (
SELECT p.lista_item
FROM pedidos_tab p
WHERE p.numpedido = 2001
)
SELECT 10, REF(s), 1, 0
FROM stock_tab s
WHERE s.numstock = 1004;

```

```

INSERT INTO THE (
SELECT p.lista_item
FROM pedidos_tab p
WHERE p.numpedido = 2001
)
VALUES (item_t(11, NULL, 2, 1));

```

En la siguiente sentencia se muestra una forma adicional de realizar la misma operación:

```

UPDATE THE (
SELECT p.lista_item
FROM pedidos_tab p
WHERE p.numpedido = 2001
) plist
SET plist.stockref =

```

```

    (SELECT REF(s)
     FROM stock_tab s
     WHERE s.numstock = 1011
    )
WHERE plist.numitem = 11;

```

3.7 Selección de valores

La siguiente sentencia invoca de forma implícita un método de comparación e ilustra cómo utiliza Oracle el método de *ordering* del objeto `pedido_t`:

```

SELECT p.numpedido
FROM pedidos_tab p
ORDER BY VALUE(p);

```

Oracle invoca al método de `MAP valor` para cada objeto de tipo `pedido_t` de la selección. Como el método simplemente devuelve el valor del atributo `numpedido`, el resultado de la selección es una lista de números de pedido ordenados en sentido ascendente.

En las siguientes líneas reproduciremos, utilizando la tecnología objeto-relaciona, las consultas de ejemplo del apartado 2.3 de la página 6:

- Cliente y datos del pedido para la orden de compra 1001:

```

SELECT Deref(p.clientref) p.dirrec_envio p.numpedido
FROM pedidos_tab p
WHERE p.numpedido = 1001;

```

- Valor total de cada pedido de compra:

```

SELECT p.numpedido, p.valor_total()
FROM pedidos_tab p;

```

- Pedidos de compra e información de la lista de items del elemento del *stock* 1004:

```

SELECT po.numpedido po.clientref.numclient,
       CURSOR (
         SELECT *
         FROM TABLE(po.lista_item) L
         WHERE L.stockref.numstock = 1004
       )
FROM pedidos_tab po;

```

3.8 Borrado de valores

El siguiente ejemplo tiene el mismo efecto que las dos operaciones de borrado necesarias en el modelo relacional (apartado 2.5 en la página 7). En este caso Oracle borra automáticamente todas las líneas de item que pertenecen al pedido borrado:

```

DELETE
FROM pedidos_tab
WHERE numpedido = 1001;

```