Un sistema de gestión de pedidos

Este ejemplo está basado en una actividad empresarial simple: gestión de pedidos de usuarios.

El ejemplo se estructura en dos partes.

- ***Aproximación relacional***: se implementa el esquema utilizando sólo los tipos de datos intrínsecos de Oracle. 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.
- ***aproximación objeto-relacional***: 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.

Entidades y relaciones

Las entidades básicas del 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*.

  1. Un pedido puede tener un número arbitrario de items, pero cada línea de la lista de *items* pertenece a un único pedido.
  2. 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*.

Entidades y relaciones...

La siguiente tabla resume la información requerida para cada una de las entidades descritas.

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 *stock*, cantidad, precio, descuento aplicable para cada item

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:

- 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 simutáneamente 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.

Aproximación relacional

- El modelo relacional normaliza las entidades y sus atributos y estructura las entidades comprador, pedidos de compra y *stock* en tablas.
- La 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.

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)
);
```

Tablas...

```
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.
- La tabla `pedidos` tiene 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.

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 Pérez', '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. Américas, 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);
```

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;
```


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;
```

Borrado

Para eliminar el pedido 1001 utilizaríamos la siguiente sentencia SQL:

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

DELETE
  FROM pedidos
 WHERE numpedido = 1001;
```

Modelo objeto-relacional

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

¿Por qué no desarrollar las aplicaciones en lenguajes de tercera generación?

Fundamentalmente por dos motivos:

- Los DBMS proporcionan funcionalidades que costaría millones de horas de trabajo replicar.
- Los lenguajes 3GL carecen de persistencia.

Esto deja al desarrollador con el problema de simular tipos complejos empleando sólo la implementación incompleta de SQL.

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.

Se produce un intenso tráfico bidireccional entre el espacio de direcciones de la aplicación y el del servidor:decremento del rendimiento.

El objetivo de la tecnología objeto-relacional es resolver estos problemas.

La vía objeto-relacional

- La aproximación objeto-relacional del ejemplo que hemos considerado comienza con las mismas entidades y relaciones esbozadas.
- Sin embargo, 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 items 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.

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.

- 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 por:
  - El orden de los números puede ser importante. Un `VARRAY` está ordenado mientras que una tabla anidada 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.
  - No existe ningún motivo especial por el cual estemos interesados en "preguntar" por la lista de números de teléfono. El uso de una tabla anidada no ofrece beneficios adicionales.

  Si el orden y los límites no son consideraciones importantes en el diseño, se debe usar la siguiente receta para decidir entre `VARRAY`s 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`.

Definición de tipos...

- La siguiente sentencia define el tipo de objeto `direccion_t` que se utilizará para representar direcciones postales.

```
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.
  - Cada objecto `cliente_t` tiene también asociado un método de clasificación, uno de los dos tipos de métodos de comparación. `ord_cliente`. La sentencia anterior no incluye el programa PL/SQL que implementa el método. Este se definirá con posterioridad.

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 onsultar 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 `VARRAY`s.
- 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.

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,
  clientref     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.

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.

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.

Definición de métodos...

- 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.
  - La misión de esta función es calcular el importe total de los elementos asociados al pedido.
  - 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. 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.

Definición de métodos...

- ...
  - 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 dereferencia implícita de `REF`s 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.

Definición de métodos...

- 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.

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 la columnas.
- Los objetos son equivalentes a las filas.

Contemplado desde esta prespectiva:

- 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.

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);
```

Dos son los aspectos relevantes que proporciona la tecnología objeto relacional:

- **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. Sin esta capacidad, sería necesario definir cada tabla de forma individual.
  - 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.

    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`.
  - Un tipo de datos abstracto puede contener atributos que son a su vez tipos de datos abstractos.
  - Cuando un tipo de dato es instanciado como un objeto, los objetos incluidos también son instanciados.

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);
```

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

La tabla de objetos `stock_tab`...

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`.

Algunas notas:

- Todas las filas de una tabla anidada se encuentran en una tabla de almacenamiento separada.
- Se utilliza 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`.

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 (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`.

Inserción de valores

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

`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);
```

`clientes_tab`

```
INSERT INTO clientes_tab
  VALUES (
    1, 'Juan Pérez',
    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')
);
```

`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 ;
```


Inserción de valores...

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

numpedido	1001
clientref	REF al cliente número 1
fechapedido	SYSDATE
fechaenvio	15 de mayo de 2001
lista_item	Una lista de items vacía
direc_envio	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 para identificar el objetivo de la inserción: una fila de la tabla anidada en la columna `lista_item` de la tabla `pedidos_tab`:

```
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.

Inserción de valores...

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));
```


Inserción de valores...

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;
```

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.

Selección de valores...

En las siguientes líneas reproduciremos, utilizando la tecnología objeto-relacional, las consultas de ejemplo que vimos en la implementación relacional:

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

```
SELECT DEREF(p.clientref) p.direc_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;
```

Borrado de valores

El siguiente ejemplo tiene el mismo efecto que las dos operaciones de borrado necesarias en el modelo relacional

En este caso Oracle borra automáticamente todas las líneas de item que pertenecen al pedido borrado:

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