```
SELECT struct(ape1: e.nombre.apellido1,
              ape2: e.nombre.apellido2,
              nom: e.nombre.nombre_pila,
              media: e.nota_media)
FROM e in estudiantes_graduados
WHERE e.tutor.nombre_pila='Gloria'
AND e.tutor.apellido1='Martínez'
ORDER BY media DESC, ape1 ASC, ape2 ASC;
```

OQL tiene además otras características que no se van a presentar aquí:

- Especificación de vistas dando nombres a consultas.
- Obtención como resultado de un solo elemento (hasta ahora hemos visto que se devuelven colecciones: `set`, `bag`, `list`).
- Uso de operadores de colecciones: funciones de agregados (`max`, `min`, `count`, `sum`, `avg`) y cuantificadores (`for all`, `exists`).
- Uso de `group by`.

5. Sistemas objeto–relacionales

El modo en que los objetos han entrado en el mundo de las bases de datos relacionales es en forma de dominios, actuando como el tipo de datos de una columna. Hay dos implicaciones muy importantes por el hecho de utilizar una clase como un dominio:

- Es posible almacenar múltiples valores en una columna de una misma fila ya que un objeto suele contener múltiples valores. Sin embargo, si se utiliza una clase como dominio de una columna, en cada fila esa columna sólo puede contener un objeto de la clase (se sigue manteniendo la restricción del modelo relacional de contener valores atómicos en la intersección de cada fila con cada columna).
- Es posible almacenar procedimientos en las relaciones porque un objeto está enlazado con el código de los procesos que sabe realizar (los métodos de su clase).

Otro modo de incoporar objetos en las bases de datos relacionales es construyendo tablas de objetos, donde cada fila es un objeto.

Ya que un sistema objeto–relacional es un sistema relacional que permite almacenar objetos en sus tablas, la base de datos sigue sujeta a las restricciones que se aplican a todas las bases de datos relacionales y conserva la capacidad de utilizar operaciones de concatenación (*join*) para implementar las relaciones "al vuelo".

5.1. Objetos en Oracle

Los tipos de objetos en Oracle son tipos de datos definidos por el usuario. La tecnología de objetos que proporciona es una capa de abstracción construida sobre su tecnología relacional, por lo que los datos se siguen almacenando en columnas y tablas. En los siguientes apartados se resume la orientación a objetos que soporta la versión 9i de Oracle.

5.1.1. Tipos de objetos y referencias

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

```
CREATE TYPE persona AS OBJECT
(
    nombre VARCHAR2(30),
    telefono VARCHAR2(20)
);

CREATE TYPE lineaped AS OBJECT
(
    nom_articulo VARCHAR2(30),
    cantidad NUMBER,
    precio_unidad NUMBER(12,2)
);

CREATE TYPE lineaped_tabla AS TABLE OF lineaped;

CREATE TYPE pedido AS OBJECT
(
    id NUMBER,
    contacto persona,
    lineasped lineaped_tabla,

    MEMBER FUNCTION obtener_valor RETURN NUMBER
);
```

`lineasped` es lo que se denomina una *tabla anidada* (*nested table*) que es un objeto de tipo colección. Una vez creados los objetos, éstos se pueden utilizar como un tipo de datos al igual que `NUMBER` o `VARCHAR2`. Por ejemplo, podemos definir una tabla relacional para guardar información de personas de contacto:

```
CREATE TABLE contactos
(
    contacto persona,
    fecha DATE
);
```

Esta es una tabla relacional que tiene una columna cuyo tipo es un objeto. Cuando los objetos se utilizan de este modo se les denomina *objetos columna*.

Cuando se declara una columna como un tipo de objeto o como una tabla anidada, se puede incluir una cláusula `DEFAULT` para asignar valores por defecto. Veamos un ejemplo:

```
CREATE TYPE persona AS OBJECT
(
    id NUMBER,
    nombre VARCHAR2(30),
    direccion VARCHAR2(30)
);

CREATE TYPE gente AS TABLE OF persona;

CREATE TABLE departamento
(
    num_dept VARCHAR2(5) PRIMARY KEY,
    nombre_dept VARCHAR2(20),
    director persona DEFAULT persona(1,'Pepe Pérez',NULL),
    empleados gente DEFAULT gente( persona(2,'Ana López','C/del Pez, 5'),
                                   persona(3,'Eva García',NULL) )
)
NESTED TABLE empleados STORE AS empleados_tab;
```

Las columnas que son tablas anidadas y los atributos que son tablas de objetos requieren una tabla a parte donde almacenar las filas de dichas tablas. Esta tabla de almacenamiento se especifica mediante la cláusula `NESTED TABLE...STORE AS...`. Para recorrer las filas de una tabla anidada se utilizan cursores anidados.

Sobre las tablas de objetos se pueden definir restricciones. En el siguiente ejemplo se muestra cómo definir una clave primaria sobre una tabla de objetos:

```
CREATE TYPE ubicacion AS OBJECT
(
    num_edificio NUMBER,
    ciudad VARCHAR2(30)
);

CREATE TYPE persona AS OBJECT
(
    id NUMBER,
    nombre VARCHAR2(30),
    direccion VARCHAR2(30),
    oficina ubicacion
);
CREATE TABLE empleados OF persona
(
    id PRIMARY KEY
);
```

El siguiente ejemplo define restricciones sobre atributos escalares de un objeto columna:

```
CREATE TABLE departamento
(
    num_dept VARCHAR2(5) PRIMARY KEY,
    nombre_dept VARCHAR2(20),
    director persona,
    despacho ubicacion,
    CONSTRAINT despacho_cons1
         UNIQUE (despacho.num_edificio,despacho.ciudad),
    CONSTRAINT despacho_cons2
         CHECK (despacho.ciudad IS NOT NULL)
);
```

Sobre las tablas de objetos también se pueden definir disparadores. Sobre las tablas de almacenamiento especificadas mediante `NESTED TABLE` no se pueden definir disparadores.

```
CREATE TABLE traslado
(
    id NUMBER,
    despacho_antiguo ubicacion,
    despacho_nuevo ubicacion
);
```

```
CREATE TRIGGER disparador
    AFTER UPDATE OF despacho ON empleados
    FOR EACH ROW
    WHEN new.despacho.ciudad='Castellon'
    BEGIN
        IF (:new.despacho.num_edificio=600) THEN
           INSERT INTO traslado (id, despacho_antiguo, despacho_nuevo)
              VALUES (:old.id, :old.despacho, :new.despacho);
        END IF;
    END;
```

Las relaciones se establecen mediante columnas o atributos `REF`. Estas relaciones pueden estar restringidas mediante la cláusula `SCOPE` o mediante una restricción de integridad referencial (`REFERENTIAL`). Cuando se restringe mediante `SCOPE`, todos lo valores almacenados en la columna `REF` apuntan a objetos de la tabla especificada en la cláusula. Sin embargo, puede ocurrir que haya valores que apunten a objetos que no existen. La restricción mediante `REFERENTIAL` es similar a la especificación de claves ajenas. La regla de integridad referencial se aplica a estas columnas, por lo que las referencias a objetos que se almacenen en estas columnas deben ser siempre de objetos que existen en la tabla referenciada.

Para evitar ambigüedades con los nombres de atributos y de métodos al utilizar la notación punto, Oracle obliga a utilizar alias para las tablas en la mayoría de las ocasiones (aunque recomienda hacerlo siempre, para evitar problemas). Por ejemplo, dadas las tablas:

```
CREATE TYPE persona AS OBJECT (dni VARCHAR2(9));
CREATE TABLE ptab1 OF persona;
CREATE TABLE ptab2 (c1 persona);
```

las siguientes consultas muestran modos correctos e incorrectos de referenciar el atributo `dni`:

```
SELECT dni FROM ptab1; -- Correcto
SELECT c1.dni FROM ptab2; -- Ilegal: notación punto sin alias de tabla
SELECT ptab2.c1.dni FROM ptab2; -- Ilegal: notación punto sin alias
SELECT p.c1.dni FROM ptab2 p; -- Correcto
```

5.1.2. Métodos

Los métodos son funciones o procedimientos que se pueden declarar en la definición de un tipo de objeto para implementar el comportamiento que se desea para dicho tipo de

objeto. Las aplicaciones llaman a los métodos para invocar su comportamiento. Para ello se utiliza también la notación punto: `objeto.metodo(lista_param)`. Aunque un método no tenga parámetros, Oracle obliga a utilizar los paréntesis en las llamadas `objeto.metodo()`. Los métodos escritos en PL/SQL o en Java, se almacenan en la base de datos. Los métodos escritos en otros lenguajes se almacenan externamente.

Hay dos clases de métodos: miembros y estáticos. Hay otro tercer tipo, los métodos constructores, que el propio sistema define para cada tipo de objeto.

Los métodos miembro son los que se utilizan para ganar acceso a los datos de una instancia de un objeto. Se debe definir un método para cada operación que se desea que haga el tipo de objeto. Estos métodos tienen un parámetro denominado `SELF` que denota a la instancia del objeto sobre la que se está invocando el método. Los métodos miembro puden hacer referencia a los atributos y a los métodos de `SELF` sin necesidad de utilizar el cualificador.

```
CREATE TYPE racional AS OBJECT
(
    num INTEGER,
    den INTEGER,
    MEMBER PROCEDURE normaliza,
    ...
);

CREATE TYPE BODY racional AS
    MEMBER PROCEDURE normaliza IS
        g INTEGER;
    BEGIN
        g := gcd(SELF.num, SELF.den);
        g := gcd(num, den); -- equivale a la línea anterior
        num := num / g;
        den := den / g;
    END normaliza;
    ...
END;
```

`SELF` no necesita declararse, aunque se puede declarar. Si no se declara, en las funciones se pasa como `IN` y en los procedimientos se pasa como `IN OUT`.

Los valores de los tipos de datos escalares siguen un orden y, por lo tanto, se pueden comparar. Sin embargo, con los tipos de objetos, que pueden tener múltiples atributos de distintos tipos, no hay un criterio predefinido de comparación. Para poder comparar objetos se debe establecer este criterio mediante métodos de mapeo o métodos de orden.

Un método de mapeo (`MAP`) permite comparar objetos mapeando instancias de objetos con tipos escalares `DATE`, `NUMBER`, `VARCHAR2` o cualquier tipo ANSI SQL como `CHARACTER` o `REAL`. Un método de mapeo es una función sin parámetros que devuelve uno de los tipos anteriores. Si un tipo de objeto define uno de estos métodos, el método se llama automáticamente para evaluar comparaciones del tipo `obj1 > obj2` y para evaluar las comparaciones que implican `DISTINCT`, `GROUP BY` y `ORDER BY`.

```
CREATE TYPE rectangulo AS OBJECT (
    alto NUMBER,
    ancho NUMBER,
    MAP MEMBER FUNCTION area RETURN NUMBER,
...
);

CREATE TYPE BODY rectangulo AS
    MAP MEMBER FUNCTION area RETURN NUMBER IS
    BEGIN
        RETURN alto*ancho;
    END area;
    ...
END;
```

Los métodos de orden `ORDER` hacen comparaciones directas objeto–objeto. Son funciones con un parámetro declarado para otro objeto del mismo tipo. El método se debe escribir para que devuelva un número negativo, cero o un número positivo, lo que significa que el objeto `SELF` es menor que, igual o mayor que el otro objeto que se pasa como parámetro. Los métodos de orden se utilizan cuando el criterio de comparación es muy complejo como para implementarlo con un método de mapeo.

Un tipo de objeto puede declarar sólo un método de mapeo o sólo un método de orden, de manera que cuando se comparan dos objetos, se llama automáticamente al método que se haya definido, sea de uno u otro tipo.

Los métodos estáticos son los que pueden ser invocados por el tipo de objeto y no por sus instancias. Estos métodos se utilizan para operaciones que son globales al tipo y que no necesitan referenciar datos de una instancia concreta. Los métodos estáticos no tienen el parámetro `SELF`. Para invocar estos métodos se utiliza la notación punto sobre el tipo del objeto: `tipo_objeto.método()`

Cada tipo de objeto tiene un método constructor implícito definido por el sistema. Este método crea un nuevo objeto (una instancia del tipo) y pone valores en sus atributos. El método constructor es una función y devuelve el nuevo objeto como su valor. El nombre del método constructor es precisamente el nombre del tipo de objeto. Sus parámetros tienen los nombres y los tipos de los atributos del tipo.

```
CREATE TABLE departamento (
    num_dept VARCHAR2(5) PRIMARY KEY,
    nombre_dept VARCHAR2(20),
    despacho ubicacion
);

INSERT INTO departamento
    VALUES ( '233', 'Ventas', ubicacion(200,'Borriol') );
```

5.1.3. Colecciones

Oracle soporta dos tipos de datos colección: las tablas anidadas y los *varray*. Un *varray* es una colección ordenada de elementos. La posición de cada elemento viene dada por un índice que permite acceder a los mismos. Cuando se define un *varray* se debe especificar el número máximo de elementos que puede contener (aunque este número se puede cambiar después). Los *varray* se almacenan como objetos opacos (`RAW` o `BLOB`). Una tabla anidada puede tener cualquier número de elementos: no se especifica ningún máximo cuando se define. Además, no se mantiene el orden de los elementos. En las tablas anidades se consultan y actualizan datos del mismo modo que se hace con las tablas relacionales. Los elementos de una tabla anidada se almacenan en una tabla a parte en la que hay una columna llamada `NESTED_TABLE_ID` que referencia a la tabla padre o al objeto al que pertenece.

```
CREATE TYPE precios AS VARRAY(10) OF NUMBER(12,2);

CREATE TYPE lineaped_tabla AS TABLE OF lineaped;
```

Cuando se utiliza una tabla anidada como una columna de una tabla o como un atributo de un objeto, es preciso especificar cuál será su tabla de almacenamiento mediante `NESTED TABLE...STORE AS...`.

Se pueden crear tipos colección multinivel, que son tipos colección cuyos elementos son colecciones.

```
CREATE TYPE satelite AS OBJECT (
    nombre VARCHAR2(20),
    diametro NUMBER );

CREATE TYPE tab_satelite AS TABLE OF satelite;
```

```
CREATE TYPE planeta AS OBJECT (
    nombre VARCHAR2(20),
    masa NUMBER,
    satelites tab_satelite );

CREATE TYPE tab_planeta AS TABLE OF planeta;
```

En este caso, la especificación de las tablas de almacenamiento se debe hacer para todas y cada una de las tablas anidadas.

```
CREATE TABLE estrellas (
    nombre VARCHAR2(20),
    edad NUMBER,
    planetas tab_planeta )
NESTED TABLE planetas STORE AS tab_alm_planetas
    (NESTED TABLE satelites STORE AS tab_alm_satelites);
```

Para crear una instancia de cualquier tipo de colección también se utiliza el método constructor, tal y como se hace con los objetos.

```
INSERT INTO estrellas
VALUES('Sol',23,
    tab_planeta(
        planeta(
            'Neptuno',
            10,
            tab_satelite(
                satelite('Proteus',67),
                satelite('Triton',82)
            )
        ),
        planeta(
            'Jupiter',
            189,
            tab_satelite(
                satelite('Calisto',97),
                satelite('Ganimedes',22)
            )
        )
    )
);
```

Las colecciones se pueden consultar con los resultados anidados:

```
SELECT e.nombre,e.proyectos
FROM empleados e;

NOMBRE   PROYECTOS
------   ---------
'Pedro'  tab_proyecto(67,82)
'Juan'   tab_proyecto(22,67,97)
```

o con los resultados sin anidar:

```
SELECT e.nombre, p.*
FROM empleados e, TABLE(e.proyectos) p;

NOMBRE   PROYECTOS
------   ---------
'Pedro'  67
'Pedro'  82
'Juan'   22
'Juan'   67
'Juan'   97
```

La notación `TABLE` sustituye a la notación `THE subconsulta` de versiones anteriores. La expresión que aparece en `TABLE` puede ser tanto el nombre de una colección como una subconsulta de una colección. Las dos consultas que se muestran a continuación obtienen el mismo resultado.

```
SELECT p.*
FROM empleados e, TABLE(e.proyectos) p
WHERE e.numemp = '18';

SELECT *
FROM TABLE(SELECT e.proyectos
           FROM empleados e
           WHERE e.numemp = '18');
```

También es posible hacer consultas con resultados no anidados sobre colecciones multinivel.

```
SELECT s.nombre
FROM estrellas e, TABLE(e.planetas) p, TABLE(p.satelites) s;
```

5.1.4. Herencia de tipos

La versión 9i es la primera versión de Oracle que soporta herencia de tipos. Cuando se crea un subtipo a partir de un tipo, el subtipo hereda todos los atributos y los métodos del tipo padre. Cualquier cambio en los atributos o métodos del tipo padre se reflejan automáticamente en el subtipo. Un subtipo se convierte en una versión especializada del tipo padre cuando al subtipo se le añaden atributos o métodos, o cuando se redefinen métodos que ha heredado, de modo que el subtipo ejecuta el método "a su manera". A esto es a lo que se denomina *polimorfismo* ya que dependiendo del tipo del objeto sobre el que se invoca el método, se ejecuta uno u otro código. Cada tipo puede heredar de un solo tipo, no de varios a la vez (no soporta herencia múltiple), pero se pueden construir jerarquías de tipos y subtipos.

Cuando se define un tipo de objeto, se determina si de él se pueden derivar subtipos mediante la cláusula `NOT FINAL`. Si no se incluye esta cláusula, se considera que es `FINAL` (no puede tener subtipos). Del mismo modo, los métodos pueden ser `FINAL` o `NOT FINAL`. Si un método es final, los subtipos no pueden redefinirlo (*override*) con una nueva implementación. Por defecto, los métodos son no finales (es decir, redefinibles).

```
CREATE TYPE t AS OBJECT ( ...,
    MEMBER PROCEDURE imprime(),
    FINAL MEMBER FUNCTION fun(x NUMBER) ...
) NOT FINAL;
```

Para crear un subtipo se utiliza la cláusula `UNDER`.

```
CREATE TYPE estudiante UNDER persona ( ...,
    titulacion VARCHAR2(30),
    fecha_ingreso DATE
) NOT FINAL;
```

El nuevo tipo, además de heredar los atributos y métodos del tipo padre, define dos nuevos atributos. A partir del subtipo se pueden derivar otros subtipos y del tipo padre tam-

bién se pueden derivar más subtipos. Para redefinir un método, se debe utilizar la cláusula `OVERRIDING`.

Los tipos y los métodos se pueden declarar como no instanciables. Si un tipo es no instanciable, no tiene método constructor, por lo que no se pueden crear instancias a partir de él. Un método no instanciable se utiliza cuando no se le va a dar una implementación en el tipo en el que se declara sino que cada subtipo va a proporcionar una implementación distinta.

```
CREATE TYPE t AS OBJECT (
    x NUMBER,
    NOT INSTANTIABLE MEMBER FUNCTION fun() RETURN NUMBER
) NOT INSTANTIABLE NOT FINAL;
```

Un tipo puede definir varios métodos con el mismo nombre pero con distinta signatura. La signatura es la combinación del nombre de un método, el número de parámetros, los tipos de éstos y el orden formal. A esto se le denomina *sobrecarga* de métodos (*overloading*).

En una jerarquía de tipos, los subtipos son variantes de la raíz. Por ejemplo, en tipo `estudiante` y el tipo `empleado` son clases de `persona`. Normalmente, cuando se trabaja con jerarquías, a veces se quiere trabajar a un nivel más general (por ejemplo, seleccionar o actualizar todas las personas) y a veces se quiere trabajar sólo son los estudiantes o sólo con los que no son estudiantes. La habilidad de poder seleccionar todas las personas juntas, pertenezcan o no a algún subtipo, es lo que se denomina *sustituibilidad*. Un súpertipo es sustituible si uno de sus subtipos puede sustituirlo en una variable, columna, etc. declarada del tipo del súpertipo. En general, los tipos son sustituibles.

- Un atributo definido como `REF miTipo` puede contener una `REF` a una instancia de `miTipo` o a una instancia de cualquier subtipo de `miTipo`.
- Un atributo definido de tipo `miTipo` puede contener una instancia de `miTipo` o una instancia de cualquier subtipo de `miTipo`.
- Una colección de elementos de tipo `miTipo` puede contener instancias de `miTipo` o instancias de cualquier subtipo de `miTipo`.

Dado el tipo `libro`:

```
CREATE TYPE libro AS OBJECT (
    titulo VARCHAR2(30),
    autor persona );
```

se puede crear una instancia de `libro` especificando un título y un autor de tipo `persona` o de cualquiera de sus subtipos, `estudiante` o `empleado`:

```
libro('BD objeto-relacionales',
    estudiante(123,'María Gil','C/Mayor,3','II','10-OCT-99')
```

A continuación se muestra un ejemplo de la sustituibilidad en las tablas de objetos.

```
CREATE TYPE persona AS OBJECT
    (id NUMBER,
    nombre VARCHAR2(30),
    direccion VARCHAR2(30)) NOT FINAL;

CREATE TYPE estudiante UNDER persona
    (titulacion VARCHAR2(10),
    especialidad VARCHAR2(30)) NOT FINAL;

CREATE TYPE estudiante_doctorado UNDER estudiante
    (programa VARCHAR2(10));

CREATE TABLE personas_tab OF persona;

INSERT INTO personas_tab
    VALUES (persona(1234,'Ana','C/Mayor,23'));

INSERT INTO personas_tab
    VALUES (estudiante(2345,'Jose','C/Paz,3','ITDI','Mecánica'));

INSERT INTO personas_tab
    VALUES (estudiante_doctorado(3456,'Luisa','C/Mar,45','IInf',NULL,'CAA'));
```

5.1.5. Funciones y predicados útiles con objetos

`VALUE` : esta función toma como parámetro un alias de tabla (de una tabla de objetos) y devuelve instancias de objetos correspondientes a las filas de la tabla.

```
SELECT VALUE(p)
FROM personas_tab p
WHERE p.direccion LIKE 'C/Mayor%';
```

La consulta devuelve todas las personas que viven en la calle Mayor, sean o no de algún subtipo.

`REF` : es una función que toma como parámetro un alias de tabla y devuelve una referencia a una instancia de un objeto de dicha tabla.

`DEREF` : es una función que devuelve la instancia del objeto correspondiente a una referencia que se le pasa como parámetro.

`IS OF` : permite formar predicados para comprobar el nivel de especialización de instancias de objetos.

```
SELECT VALUE(p)
FROM personas_tab p
WHERE VALUE(p) IS OF (estudiante);
```

De este modo se obtienen las personas que son del subtipo `estudiante` o que son de alguno de sus subtipos. Para obtener solamente aquellas personas cuyo tipo más específico es `estudiante` se utiliza la cláusula `ONLY`:

```
SELECT VALUE(p)
FROM personas_tab p
WHERE VALUE(p) IS OF (ONLY estudiante);
```

`TREAT` : es una función que trata a una instancia de un súpertipo como una instancia de uno de sus subtipos:

```
SELECT TREAT(VALUE(p) AS estudiante)
FROM personas_tab p
WHERE VALUE(p) IS OF (ONLY estudiante);
```

Bibliografía

Los capítulos 11 y 12 del texto de Elmasri y Navathe tratan ampliamente la orientación a objetos en la tecnología de bases de datos, como también se hace en los capítulos 21 y 22 del texto de Connolly, Begg y Strachan o en el capítulo 11 de Atzeni et al.. Este es un tema que aparece en gran parte de los textos básicos sobre bases de datos, aunque sólo los más recientes tratan los sistemas objeto–relacionales. El texto de Elmasri y Navathe los analiza en el capítulo 13, mientras que el texto de Connolly, Begg y Strachan lo hace en el capítulo 23 (Atzeni et al. los trata en el mismo capítulo 11).