CAPÍTULO 9

Bases de datos basadas en objetos

Las aplicaciones tradicionales de las bases de datos consisten en tareas de procesamiento de datos, como la gestión bancaria y de nóminas, con tipos de datos relativamente sencillos, que se adaptan bien al modelo relacional. A medida que los sistemas de bases de datos se fueron aplicando a un rango más amplio de aplicaciones, como el diseño asistido por computadora y los sistemas de información geográfica, las limitaciones impuestas por el modelo relacional se convirtieron en un obstáculo. La solución fue la introducción de bases de datos basadas en objetos, que permiten trabajar con tipos de datos complejos.

9.1 Visión general

El primer obstáculo al que se enfrentan los programadores que usan el modelo relacional de datos es el limitado sistema de tipos soportado por el modelo relacional. Los dominios de aplicación complejos necesitan tipos de datos del mismo nivel de complejidad, como las estructuras de registros anidados, los atributos multivalorados y la herencia, que los lenguajes de programación tradicionales soportan. La notación E-R y las notaciones E-R extendidas soportan, de hecho, estas características, pero hay que trasladarlas a tipos de datos de SQL más sencillos. El **modelo de datos relacional orientado a objetos** extiende el modelo de datos relacional ofreciendo un sistema de tipos más rico que incluye tipos de datos complejos y orientación a objetos. Hay que extender de manera acorde los lenguajes de consultas relacionales, en especial SQL, para que puedan trabajar con este sistema de tipos más rico. Estas extensiones intentan conservar los fundamentos relacionales—en especial, el acceso declarativo a los datos—mientras extienden la potencia de modelado. Los **sistemas de bases de datos relacionales basadas en objetos**, es decir, los sistemas de bases de datos basados en el modelo objeto-relación, ofrecen un medio de migración cómodo para los usuarios de las bases de datos relacionales que deseen usar características orientadas a objetos.

El segundo obstáculo es la dificultad de acceso a los datos de la base de datos desde los programas escritos en lenguajes de programación como C++ o Java. La mera extensión del sistema de tipos soportado por las bases de datos no resulta suficiente para resolver completamente este problema. Las diferencias entre el sistema de tipos de las bases de datos y el de los lenguajes de programación hace más complicados el almacenamiento y la recuperación de los datos, y se debe minimizar. Tener que expresar el acceso a las bases de datos mediante un lenguaje (SQL) que es diferente del lenguaje de programación también hace más difícil el trabajo del programador. Es deseable, para muchas aplicaciones, contar con estructuras o extensiones del lenguaje de programación que permitan el acceso directo a los datos de la base de datos, sin tener que pasar por un lenguaje intermedio como SQL.

El término **lenguajes de programación persistentes** hace referencia a las extensiones de los lenguajes de programación existentes que añaden persistencia y otras características de las bases de datos usando el sistema de tipos nativo del lenguaje de programación. El término **sistemas de bases de datos orientadas a objetos** se usa para hacer referencia a los sistemas de bases de datos que soportan sistemas de

tipos orientados a objetos y permiten el acceso directo a los datos desde los lenguajes de programación orientados a objetos usando el sistema de tipos nativo del lenguaje.

En este capítulo se explicará primero el motivo del desarrollo de los tipos de datos complejos. Luego se estudiarán los sistemas de bases de datos relacionales orientados a objetos; el tratamiento se basará en las extensiones relacionales orientadas a objetos añadidas a la versión SQL:1999 de la norma de SQL. La descripción se basa en la norma de SQL, concretamente, en el uso de características que se introdujeron en SQL:1999 y en SQL:2003. Téngase en cuenta que la mayor parte de los productos de bases de datos sólo soportan un subconjunto de las características de SQL aquí descritas. Se debe consultar el manual de usuario del sistema de bases de datos que se utilice para averiguar las características que soporta.

Luego se estudian brevemente los sistemas de bases de datos orientados a objetos que añaden el soporte de la persistencia a los lenguajes de programación orientados a objetos. Finalmente, se describen situaciones en las que el enfoque relacional orientado a objetos es mejor que el enfoque orientado a objetos, y viceversa, y se mencionan criterios para escoger entre los dos.

9.2 Tipos de datos complejos

Las aplicaciones de bases de datos tradicionales consisten en tareas de procesamiento de datos, tales como la banca y la gestión de nóminas. Dichas aplicaciones presentan conceptualmente tipos de datos simples. Los elementos de datos básicos son registros bastante pequeños y cuyos campos son atómicos, es decir, no contienen estructuras adicionales y en los que se cumple la primera forma normal (véase el Capítulo 7). Además, sólo hay unos pocos tipos de registros.

En los últimos años, ha crecido la demanda de formas de abordar tipos de datos más complejos. Considérense, por ejemplo, las direcciones. Mientras que una dirección completa se puede considerar como un elemento de datos atómico del tipo cadena de caracteres, esa forma de verlo esconde detalles como la calle, la población, la provincia, y el código postal, que pueden ser interesantes para las consultas. Por otra parte, si una dirección se representa dividiéndola en sus componentes (calle, población, provincia y código postal) la escritura de las consultas sería más complicada, pues tendrían que mencionar cada campo. Una alternativa mejor es permitir tipos de datos estructurados, que admiten el tipo *dirección* con las subpartes *calle, población, provincia* y *código_postal*.

Como ejemplo adicional, considérense los atributos multivalorados del modelo E-R. Esos atributos resultan naturales, por ejemplo, para la representación de números de teléfono, ya que las personas pueden tener más de un teléfono. La alternativa de la normalización mediante la creación de una nueva relación resulta costosa y artificial para este ejemplo.

Con sistemas de tipos complejos se pueden representar directamente conceptos del modelo E-R, como los atributos compuestos, los atributos multivalorados, la generalización y la especialización, sin necesidad de una compleja traducción al modelo relacional.

En el Capítulo 7 se definió la *primera forma normal* (1FN), (1FN) que exige que todos los atributos tengan *dominios atómicos*. Recuérdese que un dominio es *atómico* si se considera que los elementos del dominio son unidades indivisibles.

La suposición de la 1FN es natural en los ejemplos bancarios que se han considerado. No obstante, no todas las aplicaciones se modelan mejor mediante relaciones en la 1FN. Por ejemplo, en vez de considerar la base de datos como un conjunto de registros, los usuarios de ciertas aplicaciones la ven como un conjunto de objetos (o de entidades). Puede que cada objeto necesite varios registros para su representación. Una interfaz sencilla y fácil de usar necesita una correspondencia de uno a uno entre el concepto intuitivo de objeto del usuario y el concepto de elemento de datos de la base de datos.

Considérese, por ejemplo, una aplicación para una biblioteca y supóngase que se desea almacenar la información siguiente para cada libro:

- Título del libro
- Lista de autores
- Editor
- Conjunto de palabras clave

Es evidente que, si se define una relación para esta información varios dominios no son atómicos.

- **Autores**. Cada libro puede tener una lista de autores, que se pueden representar como array. Pese a todo, puede que se desee averiguar todos los libros de los que es autor Santos. Por tanto, lo que interesa es una subparte del elemento del dominio "autores".
- **Palabras clave**. Si se almacena un conjunto de palabras clave para cada libro, se espera poder recuperar todos los libros cuyas palabras clave incluyan uno o más términos dados. Por tanto, se considera el dominio del conjunto de palabras clave como no atómico.
- **Editor**. A diferencia de *palabras clave* y *autores*, *editor* no tiene un dominio que se evalúe en forma de conjunto. No obstante, se puede considerar que *editor* consta de los subcampos *nombre* y *sucursal*. Este punto de vista hace que el dominio de *editor* no sea atómico.

La Figura 9.1 muestra una relación de ejemplo, *libros*.

Por simplificar se da por supuesto que el título del libro lo identifica de manera unívoca[1]. Se puede representar, entonces, la misma información usando el esquema siguiente:

- *autores*(*título, autor, posición*)
- *palabras_clave*(*título, palabra_clave*)
- *libros4*(*título, nombre_editor, sucursal_editor*)

El esquema anterior satisface la 4NF. La Figura 9.2 muestra la representación normalizada de los datos de la Figura 9.1.

Aunque la base de datos de libros de ejemplo puede expresarse de manera adecuada sin necesidad de usar relaciones anidadas, su uso lleva a un modelo más fácil de comprender. El usuario típico o el programador de sistemas de recuperación de la información piensa en la base de datos en términos de libros que tienen conjuntos de autores, como los modelos de diseño que no se hallan en la 1FN. El diseño de la 4FN exige consultas que reúnan varias relaciones, mientras que los diseños que no se hallan en la 1FN hacen más fáciles muchos tipos de consultas.

Por otro lado, en otras situaciones puede resultar más conveniente usar una representación en la primera forma normal en lugar de conjuntos. Por ejemplo, considérese la relación *impositor* del ejemplo bancario. La relación es de varios a varios entre *clientes* y *cuentas*. Teóricamente, se podría almacenar un conjunto de cuentas con cada cliente, o un conjunto de clientes con cada cuenta, o ambas cosas. Si se almacenaran ambas cosas, se tendría redundancia de los datos (la relación de un cliente concreto con una cuenta dada se almacenaría dos veces).

La posibilidad de usar tipos de datos complejos como los conjuntos y los arrays puede resultar útil en muchas aplicaciones, pero se debe usar con cuidado.

9.3 Tipos estructurados y herencia en SQL

Antes de SQL:1999 el sistema de tipos de SQL consistía en un conjunto bastante sencillo de tipos predefinidos. SQL:1999 añadió un sistema de tipos extenso a SQL, lo que permite los tipos estructurados y la herencia de tipos.

1. Esta suposición no se cumple en el mundo real. Los libros se suelen identificar por un número de ISBN de diez cifras que identifica de manera unívoca cada libro publicado.

título	*array_autores*	*editor*	*conjunto_palabras_clave*
		(*nombre, sucursal*)	
Compiladores	[Gómez, Santos]	(McGraw-Hill, NuevaYork)	{análisis sintáctico, análisis}
Redes	[Santos, Escudero]	(Oxford, Londres)	{Internet, Web}

Figura 9.1 Relación de libros que no está en la 1FN, *libros*.

título	*autor*	*posición*
Compiladores	Gómez	1
Compiladores	Santos	2
Redes	Santos	1
Redes	Escudero	2

autores

título	*palabra_clave*
Compiladores	análisis sintáctico
Compiladores	análisis
Redes	Internet
Redes	Web

palabras_clave

título	*nombre_editor*	*sucursal_editor*
Compiladores	McGraw-Hill	Nueva York
Redes	Oxford	Londres

libros4

Figura 9.2 Versión en la 4FN de la relación *libros*.

9.3.1 Tipos estructurados

Los tipos estructurados permiten representar directamente los atributos compuestos de los diagramas E-R. Por ejemplo, se puede definir el siguiente tipo estructurado para representar el atributo compuesto *nombre* con los atributos componentes *nombredepila* y *apellidos*:

```
create type Nombre as
    (nombredepila varchar(20),
    apellidos varchar(20))
    final
```

De manera parecida, el tipo estructurado siguiente puede usarse para representar el atributo compuesto *dirección*:

```
create type Dirección as
    (calle varchar(20),
    ciudad varchar(20),
    códigopostal varchar(9))
    not final
```

En SQL estos tipos se denominan tipos **definidos por el usuario**. La definición anterior corresponde al diagrama E-R de la Figura 6.4. Las especificaciones **final** y **not final** están relacionadas con la subtipificación, que se describirá más adelante, en el Apartado 9.3.22.[2] Ahora se pueden usar esos tipos para crear atributos compuestos en las relaciones, con sólo declarar que un atributo es de uno de estos tipos. Por ejemplo, se puede crear la tabla *cliente* de la manera siguiente:

```
create table cliente (
    nombre Nombre,
    dirección Dirección,
    fechaDeNacimiento date)
```

2. La especificación **final** de *Nombre* indica que no se pueden crear subtipos de *nombre*, mientras que la especificación **not final** de *Dirección* indica que se pueden crear subtipos de *dirección*.

Se puede tener acceso a los componentes de los atributos compuestos usando la notación "punto"; por ejemplo, *nombre.nombredepila* devuelve el componente nombre de pila del atributo nombre. El acceso al atributo *nombre* devolvería un valor del tipo estructurado *Nombre*.

También se puede crear una tabla cuyas filas sean de un tipo definido por el usuario. Por ejemplo, se puede definir el tipo *TipoCliente* y crear la tabla *cliente* de la manera siguiente:

```
create type TipoCliente as (
        nombre Nombre,
        dirección Dirección,
        fechaDeNacimiento date)
        not final
create table cliente of TipoCliente
```

Una manera alternativa de definir los atributos compuestos en SQL es usar **tipos de fila** sin nombre. Por ejemplo, la relación que representa la información del cliente se podría haber creado usando tipos de fila de la manera siguiente:

```
create table cliente (
    nombre row (nombredepila varchar(20),
                apellidos varchar(20)),
    dirección row (calle varchar(20),
                   ciudad varchar(20),
                   códigopostal varchar(9)),
    fechaDeNacimiento date)
```

Esta definición es equivalente a la anterior definición de la tabla, salvo en que los atributos *nombre* y *dirección* tienen tipos sin nombre y las filas de la tabla también tienen un tipo sin nombre.

La consulta siguiente ilustra la manera de tener acceso a los atributos componentes de los atributos compuestos. La consulta busca el apellido y la ciudad de cada cliente.

```
select nombre.apellidos, dirección.ciudad
from cliente
```

Sobre los tipos estructurados se pueden definir **métodos**. Los métodos se declaran como parte de la definición de los tipos de los tipos estructurados:

```
create type TipoCliente as (
        nombre Nombre,
        dirección Dirección,
        fechaDeNacimiento date)
        not final
    method edadAFecha(aFecha date)
        returns interval year
```

El cuerpo del método se crea por separado:

```
create instance method edadAFecha (aFecha date)
    returns interval year
    for TipoCliente
begin
    return aFecha − self.fechaDeNacimiento;
end
```

Téngase en cuenta que la cláusula **for** indica el tipo al que se aplica el método, mientras que la palabra clave **instance** indica que el método se ejecuta sobre un ejemplar del tipo *Cliente*. La variable **self** hace referencia al ejemplar de *Cliente* sobre el que se invoca el método. El cuerpo del método puede contener

instrucciones procedimentales, que ya se han visto en el Apartado 4.6. Los métodos pueden actualizar los atributos del ejemplar sobre el que se ejecutan.

Los métodos se pueden invocar sobre los ejemplares de los tipos. Si se hubiera creado la tabla *cliente* del tipo *TipoCliente*, se podría invocar el método *edadAFecha* () como se ilustra a continuación, para averiguar la edad de cada cliente.

```
select nombre.apellidos, edadAFecha(current_date)
from cliente
```

En SQL:1999 se usan **funciones constructoras** para crear valores de los tipos estructurados. Las funciones con el mismo nombre que un tipo estructurado son funciones constructoras de ese tipo estructurado. Por ejemplo, se puede declarar una función constructora para el tipo *Nombre* de esta manera:

```
create function Nombre (nombredepila varchar(20), apellidos varchar(20))
returns Nombre
begin
    set self.nombredepila = nombredepila;
    set self.apellidos = apellidos;
end
```

Se puede usar **new** *Nombre* ('Martín', 'Gómez') para crear un valor del tipo *Nombre*.

Se puede crear un valor de fila haciendo una relación de sus atributos entre paréntesis. Por ejemplo, si se declara el atributo *nombre* como tipo de fila con los componentes *nombredepila* y *apellidos*, se puede crear para él este valor:

```
('Ted', 'Codd')
```

sin necesidad de función constructora.

De manera predeterminada, cada tipo estructurado tiene una función constructora sin argumentos, que configura los atributos con sus valores predeterminados. Cualquier otra función constructora hay que crearla de manera explícita. Puede haber más de una función constructora para el mismo tipo estructurado; aunque tengan el mismo nombre, deben poder distinguirse por el número y tipo de sus argumentos.

La instrucción siguiente ilustra la manera de crear una nueva tupla de la relación *Cliente*. Se da por supuesto que se ha definido una función constructora para *Dirección*, igual que la función constructora que se definió para *Nombre*.

```
insert into Cliente
    values
        (new Nombre('Martín', 'Gómez'),
        new Dirección('Calle Mayor, 20', 'Madrid', '28045'),
        date '22-8-1960')
```

9.3.2 Herencia de tipos

Supóngase que se tiene la siguiente definición de tipo para las personas:

```
create type Persona
    (nombre varchar(20),
     dirección varchar(20))
```

Puede que se desee almacenar en la base de datos información adicional sobre las personas que son estudiantes y sobre las que son profesores. Dado que los estudiantes y los profesores también son personas, se puede usar la herencia para definir en SQL los tipos estudiante y profesor:

```
create type Estudiante
    under Persona
    (grado varchar(20),
     departmento varchar(20))
create type Profesor
    under Persona
    (sueldo integer,
     departmento varchar(20))
```

Tanto *Estudiante* como *Profesor* heredan los atributos de *Persona*—es decir, *nombre* y *dirección*. Se dice que *Estudiante* y *Profesor* son subtipos de *Persona*, y *Persona* es un supertipo de *Estudiante* y de *Profesor*.

Los métodos de los tipos estructurados se heredan por sus subtipos, igual que los atributos. Sin embargo, cada subtipo puede redefinir el efecto de los métodos volviendo a declararlos, usando **overriding method** en lugar de **method** en la declaración del método.

La norma de SQL también exige un campo adicional al final de la definición de los tipos, cuyo valor es **final** o **not final**. La palabra clave **final** indica que no se pueden crear subtipos a partir del tipo dado, mientras que **not final** indica que se pueden crear subtipos. Ahora, supóngase que se desea almacenar información sobre los profesores ayudantes, que son a la vez estudiantes y profesores, quizás incluso en departamentos diferentes. Esto se puede hacer si el sistema de tipos soporta **herencia múltiple**, por la que se pueden declarar los tipos como subtipos de varios tipos. Téngase en cuenta que la norma de SQL (hasta las versiones SQL:1999 y SQL:2003, al menos) no soporta la herencia múltiple, aunque puede que sí que la soporten versiones futuras.

Por ejemplo, si el sistema de tipos soporta la herencia múltiple, se puede definir el tipo para los profesores ayudantes de la manera siguiente:

```
create type ProfesorAyudante
    under Estudiante, Profesor
```

ProfesorAyudante heredará todos los atributos de *Estudiante* y de *Profesor*. No obstante, hay un problema, ya que los atributos *nombre*, *dirección* y *departamento* están presentes tanto en *Estudiante* como en *Profesor*.

Los atributos *nombre* y *dirección* se heredan realmente de una fuente común, *Persona*. Por tanto, no hay ningún conflicto causado por heredarlos de *Estudiante* y de *Profesor*. Sin embargo, el atributo *departamento* se define por separado en *Estudiante* y en *Profesor*. De hecho, cada profesor ayudante puede ser estudiante en un departamento y profesor en otro. Para evitar conflictos entre las dos apariciones de *departamento*, se pueden rebautizar usando una cláusula **as**, como en la definición del tipo *ProfesorAyudante*:

```
create type ProfesorAyudante
    under Estudiante with (departmento as departamento_estudiante),
          Profesor with (departmento as departamento_profesor)
```

Hay que tener en cuenta nuevamente que SQL sólo soporta la herencia simple—es decir, cada tipo sólo se puede heredar de un único tipo, la sintaxis usada es como la de los ejemplos anteriores. La herencia múltiple, como en el ejemplo de *ProfesorAyudante*, *no* está soportada en SQL.

En SQL, como en la mayor parte del resto de los lenguajes, el valor de un tipo estructurado debe tener exactamente un "tipo más concreto". Es decir, cada valor debe asociarse, al crearlo, con un tipo concreto, denominado su **tipo más concreto**. Mediante la herencia también se asocia con cada uno de los supertipos de su tipo más concreto. Por ejemplo, supóngase que una entidad es tanto del tipo *Persona* como del tipo *Estudiante*. Entonces, el tipo más concreto de la entidad es *Estudiante*, ya que *Estudiante* es un subtipo de *Persona*. Sin embargo, una entidad no puede ser de los tipos *Estudiante* y *Profesor*, a menos que tenga un tipo, como *ProfesorAyudante*, que sea subtipo de *Profesor* y de *Estudiante* (lo cual no es posible en SQL, ya que SQL no soporta la herencia múltiple).

9.4 Herencia de tablas

Las subtablas de SQL se corresponden con el concepto de especialización/generalización de E-R. Por ejemplo, supóngase que se define la tabla *personas* de la manera siguiente:

create table *personas* **of** *Persona*

A continuación se pueden definir las tablas *estudiantes* y *profesores* como **subtablas** de *personas*, de la manera siguiente:

create table *estudiantes* **of** *Estudiante*
under *personas*
create table *profesores* **of** *Profesor*
under *personas*

Los tipos de las subtablas deben ser subtipos del tipo de la tabla madre. Por tanto, todos los atributos presentes en *personas* también están presentes en las subtablas.

Además, cuando se declaran *estudiantes* y *profesores* como subtablas de *personas*, todas las tuplas presentes en *estudiantes* o en *profesores* pasan a estar también presentes de manera implícita en *personas*. Por tanto, si una consulta usa la tabla *personas*, no sólo encuentra tuplas directamente insertadas en esa tabla, sino también tuplas insertadas en sus subtablas, es decir, *estudiantes* y *profesores*. No obstante, esa consulta sólo puede tener acceso a los atributos que están presentes en *personas*.

SQL permite hallar tuplas que se encuentran en *personas* pero no en sus subtablas usando en las consultas "**only** *personas*" en lugar de *personas*. La palabra clave **only** también puede usarse en las sentencias delete y update. Sin la palabra clave **only**, la instrucción delete aplicada a una supertabla, como *personas*, también borra las tuplas que se insertaron originalmente en las subtablas (como *estudiantes*); por ejemplo, la instrucción

delete from *personas* **where** *P*

borrará todas las tuplas de la tabla *personas*, así como de sus subtablas *estudiantes* y *profesores*, que satisfagan *P*. Si se añade la palabra clave **only** a la instrucción anterior, las tuplas que se insertaron en las subtablas no se ven afectadas, aunque satisfagan las condiciones de la cláusula **where**. Las consultas posteriores a la supertabla seguirán encontrando esas tuplas.

Teóricamente, la herencia múltiple es posible con las tablas, igual que con los tipos. Por ejemplo, se puede crear una tabla del tipo *ProfesorAyudante*:

create table *profesores_ayudantes*
of *ProfesorAyudante*
under *estudiantes*, *profesores*

Como consecuencia de la declaración, todas las tuplas presentes en la tabla *profesores_ayudantes* también se hallan presentes de manera implícita en las tablas *profesores* y *estudiantes* y, a su vez, en la tabla *personas*. Hay que tener en cuenta, no obstante, que SQL no soporta la herencia múltiple de tablas.

Existen varios requisitos de consistencia para las subtablas. Antes de definir las restricciones, es necesaria una definición: se dice que las tuplas de una subtabla se **corresponden** con las tuplas de la tabla madre si tienen el mismo valor para todos los atributos heredados. Por tanto, las tuplas correspondientes representan a la misma entidad.

Los requisitos de consistencia de las subtablas son:

1. Cada tupla de la supertabla puede corresponderse, como máximo, con una tupla de cada una de sus subtablas inmediatas.
2. SQL posee una restricción adicional que hace que todas las tuplas que se corresponden entre sí deben proceder de una tupla (insertada en una tabla).

Por ejemplo, sin la primera condición, se podrían tener dos tuplas de *estudiantes* (o de *profesores*) que correspondieran a la misma persona.

La segunda condición excluye que haya una tupla de *personas* correspondiente a una tupla de *estudiantes* y a una tupla de *profesores*, a menos que todas esas tuplas se hallen presentes de manera implícita porque se haya insertado una tupla en la tabla *profesores_ayudantes*, que es subtabla tanto de *profesores* como de *estudiantes*.

Dado que SQL no soporta la herencia múltiple, la segunda condición impide realmente que ninguna persona sea a la vez profesor y estudiante. Aunque se soportara la herencia múltiple, surgiría el mismo problema si estuviera ausente la subtabla *profesores_ayudantes*. Evidentemente, resultaría útil modelar una situación en la que alguien pudiera ser a la vez profesor y estudiante, aunque no se hallara presente la subtabla *profesores_ayudantes*. Por tanto, puede resultar útil eliminar la segunda restricción de consistencia. Hacerlo permitiría que cada objeto tuviera varios tipos, sin necesidad de que tuviera un tipo más concreto.

Por ejemplo, supóngase que se vuelve a tener el tipo *Persona*, con los subtipos *Estudiante* y *Profesor*, y la tabla correspondiente *personas*, con las subtablas *profesores* y *estudiantes*. Se puede tener una tupla de *profesores* y una tupla de *estudiantes* correspondientes a la misma tupla de *personas*. No hace falta tener el tipo *ProfesorAyudante*, que es subtipo de *Estudiante* y de *Profesor*. No hace falta crear el tipo *ProfesorAyudante* a menos que se desee almacenar atributos adicionales o redefinir los métodos de manera específica para las personas que sean a la vez estudiantes y profesores.

No obstante, hay que tener en cuenta que, por desgracia, SQL prohíbe las situaciones de este tipo debido al segundo requisito de consistencia. Ya que SQL tampoco soporta la herencia múltiple, no se puede usar la herencia para modelar una situación en la que alguien pueda ser a la vez estudiante y profesor. En consecuencia, las subtablas de SQL no se pueden usar para representar las especializaciones que se solapen de los modelos E-R.

Por supuesto, se pueden crear tablas diferentes para representar las especializaciones o generalizaciones que se solapen sin usar la herencia. El proceso ya se ha descrito anteriormente, en el Apartado 6.9.5. En el ejemplo anterior, se crearían las tablas *personas*, *estudiantes* y *profesores*, de las que las tablas *estudiantes* y *profesores* contendrían el atributo de clave primaria de *Persona* y otros atributos específicos de *Estudiante* y de *Profesor*, respectivamente. La tabla *personas* contendría la información sobre todas las personas, incluidos los estudiantes y los profesores. Luego habría que añadir las restricciones de integridad referencial correspondientes para garantizar que los estudiantes y los profesores también se hallen representados en la tabla *personas*.

En otras palabras, se puede crear una implementación mejorada del mecanismo de las subtablas mediante las características de SQL ya existentes, con algún esfuerzo adicional para la definición de la tabla, así como en el momento de las consultas para especificar las reuniones para el acceso a los atributos necesarios.

Para finalizar este apartado, hay que tener en cuenta que SQL define un nuevo privilegio denominado **under**, el cual es necesario para crear subtipos o subtablas bajo otro tipo o tabla. La razón de ser de este privilegio es parecida a la del privilegio **references**.

9.5 Tipos array y multiconjunto en SQL

SQL soporta dos tipos de conjuntos: arrays y multiconjuntos; los tipos array se añadieron en SQL:1999, mientras que los tipos multiconjunto se agregaron en SQL:2003. Recuérdese que un *multiconjunto* es un conjunto no ordenado, en el que cada elemento puede aparecer varias veces. Los multiconjuntos son como los conjuntos, salvo que los conjuntos permiten que cada elemento aparezca, como mucho, una vez.

Supóngase que se desea registrar información sobre libros, incluido un conjunto de palabras clave para cada libro. Supóngase también que se deseara almacenar el nombre de los autores de un libro en forma de array; a diferencia de los elementos de los multiconjuntos, los elementos de los arrays están ordenados, de modo que se puede distinguir el primer autor del segundo, etc. El ejemplo siguiente ilustra la manera en que se pueden definir en SQL estos atributos valorados como arrays y como multiconjuntos.

```
create type Editor as
      (nombre varchar(20),
       sucursal varchar(20))
create type Libro as
      (título varchar(20),
       array_autores varchar(20) array [10],
       fecha_publicación date,
       editor Editor,
       conjunto_palabras_clave varchar(20) multiset)
create table libros of Libro
```

La primera instrucción define el tipo denominado *Editor*, que tiene dos componentes: nombre y sucursal. La segunda instrucción define el tipo estructurado *Libro*, que contiene un *título*, un *array_autores*, que es un array de hasta diez nombres de autor, una fecha de publicación, un editor (del tipo *Editor*), y un multiconjunto de palabras clave. Finalmente, se crea la tabla *libros*, que contiene las tuplas del tipo *Libro*.

Téngase en cuenta que se ha usado un array, en lugar de un multiconjunto, para almacenar el nombre de los autores, ya que el orden de los autores suele tener cierta importancia, mientras que se considera que el orden de las palabras asociadas con el libro no es significativo.

En general, los atributos multivalorados de los esquemas E-R se pueden asignar en SQL a atributos valorados como multiconjuntos; si el orden es importante, se pueden usar los arrays de SQL en lugar de los multiconjuntos.

9.5.1 Creación y acceso a los valores de los conjuntos

En SQL:1999 se puede crear un array de valores de esta manera:

array['Silberschatz', 'Korth', 'Sudarshan']

De manera parecida, se puede crear un multiconjunto de palabras clave de la manera siguiente:

multiset['computadora', 'base de datos', 'SQL']

Por tanto, se puede crear una tupla del tipo definido por la relación *libros* como:

('Compiladores', **array**['Gómez', 'Santos'], **new** *Editor*('McGraw-Hill', 'Nueva York'),
multiset['análisis sintáctico', 'análisis'])

En este ejemplo se ha creado un valor para el atributo *Editor* mediante la invocación a una función *constructora* para *Editor* con los argumentos correspondientes. Téngase en cuenta que esta constructora para *Editor* se debe crear de manera explícita y no se halla presente de manera predeterminada; se puede declarar como la constructora para *Nombre*, que ya se ha visto en el Apartado 9.3.

Si se desea insertar la tupla anterior en la relación *libros*, se puede ejecutar la instrucción:

```
insert into libros
values
('Compiladores', array['Gómez', 'Santos'],
                 new Editor('McGraw-Hill', 'Nueva York'),
                 multiset['análisis sintáctico', 'análisis'])
```

Se puede tener acceso a los elementos del array o actualizarlos especificando el índice del array, por ejemplo, *array_autores* [1].

9.5.2 Consulta de los atributos valorados como conjuntos

Ahora se considerará la forma de manejar los atributos que se valoran como conjuntos. Las expresiones que se valoran como conjuntos pueden aparecer en cualquier parte en la que pueda aparecer el nombre

de una relación, como las cláusulas **from**, como ilustran los siguientes párrafos. Se usará la tabla *libros* que ya se había definido anteriormente.

Si se desea averiguar todos los libros que tienen las palabras "base de datos" entre sus palabras clave se puede usar la consulta siguiente:

```
select título
from libros
where 'base de datos' in (unnest(conjunto_palabras_clave))
```

Obsérvese que se ha usado **unnest**(*conjunto_palabras_clave*) en una posición en la que SQL sin las relaciones anidadas habría exigido una subexpresión **select-from-where**.

Si se sabe que un libro concreto tiene tres autores, se puede escribir:

```
select array_autores[1], array_autores[2], array_autores[3]
from libros
where título = 'Fundamentos de bases de datos'
```

Ahora supóngase que se desea una relación que contenga parejas de la forma "título, nombre_autor" para cada libro y para cada uno de sus autores. Se puede usar esta consulta:

```
select L.título, A.autores
from libros as L, unnest(L.array_autores) as A(autores)
```

Dado que el atributo *array_autores* de *libros* es un campo que se valora como conjunto, **unnest**(*L.array_autores*) puede usarse en una cláusula **from** en la que se espere una relación. Téngase en cuenta que la variable tupla *B* es visible para esta expresión, ya que se ha definido *anteriormente* en la cláusula **from**.

Al desanidar un array la consulta anterior pierde información sobre el orden de los elementos del array. Se puede usar la cláusula **unnest with ordinality** para obtener esta información, como ilustra la consulta siguiente. Esta consulta se puede usar para generar la relación *autores*, que se ha visto anteriormente, a partir de la relación *libros*.

```
select título, A.autores, A.posición
from libros as L,
        unnest(L.array_autores) with ordinality as A(autores, posición)
```

La cláusula **with ordinality** genera un atributo adicional que registra la posición del elemento en el array. Se puede usar una consulta parecida, pero sin la cláusula **with ordinality**, para generar la relación *palabras_clave*.

9.5.3 Anidamiento y desanidamiento

La transformación de una relación anidada en una forma con menos atributos de tipo relación (o sin ellos) se denomina **desanidamiento**. La relación *libros* tiene dos atributos, *array_autores* y *conjunto_palabras_clave*, que son conjuntos, y otros dos, *título* y *editor*, que no lo son. Supóngase que se desea convertir la relación en una sola relación plana, sin relaciones anidadas ni tipos estructurados como atributos. Se puede usar la siguiente consulta para llevar a cabo la tarea:

```
select título, A.autor, editor.nombre as nombre_editor, editor.sucursal
        as sucursal_editor, P.palabras_clave
from libros as L, unnest(L.array_autores) as A(autores),
        unnest (L.conjunto_palabras_clave) as P(palabras_clave)
```

La variable *L* de la cláusula **from** se declara para que tome valores de *libros*. La variable *A* se declara para que tome valores de los autores de *array_autores* para el libro *L* y *P* se declara para que tome valores de las palabras clave del *conjunto_palabras_clave* del libro *L*. La Figura 9.1 muestra un ejemplar de la relación *libros* y la Figura 9.3 muestra la relación, denominada *libros_plana*, que es resultado de la consulta

título	*autor*	*nombre_editor*	*sucursal_editor*	*conjunto_palabras_clave*
Compiladores	Gómez	McGraw-Hill	Nueva York	análisis sintáctico
Compiladores	Santos	McGraw-Hill	Nueva York	análisis sintáctico
Compiladores	Gómez	McGraw-Hill	Nueva York	análisis
Compiladores	Santos	McGraw-Hill	Nueva York	análisis
Redes	Santos	Oxford	Londres	Internet
Redes	Escudero	Oxford	Londres	Internet
Redes	Santos	Oxford	Londres	Web
Redes	Escudero	Oxford	Londres	Web

Figura 9.3 *libros_plana*: resultado del desanidamiento de los atributos *array_autores* y *conjunto_palabras_clave* de la relación *libros*.

anterior. Téngase en cuenta que la relación *libros_plana* se halla en 1FN, ya que todos sus atributos son atómicos.

El proceso inverso de transformar una relación en la 1FN en una relación anidada se denomina **anidamiento**. El anidamiento puede realizarse mediante una extensión de la agrupación en SQL. En el uso normal de la agrupación en SQL se crea (lógicamente) una relación multiconjunto temporal para cada grupo y se aplica una función de agregación a esa relación temporal para obtener un valor único (atómico). La función **collect** devuelve el multiconjunto de valores; en lugar de crear un solo valor se puede crear una relación anidada. Supóngase que se tiene la relación en 1FN *libros_plana*, tal y como se muestra en la Figura 9.3. La consulta siguiente anida la relación en el atributo *palabras_clave*:

select *título*, *autores*, *Editor*(*nombre_editor, sucursal_editor*) **as** *editor*,
collect(*palabras_clave*) **as** *conjunto_palabras_clave*
from *libros_plana*
group by *título*, *autor*, *editor*

El resultado de la consulta a la relación *libros_plana* de la Figura 9.3 aparece en la Figura 9.4.

Si se desea anidar también el atributo autores en un multiconjunto, se puede usar la consulta:

select *título*, **collect**(*autores*) **as** *conjunto_autores*,
Editor(*nombre_editor, sucursal_editor*) **as** *editor*,
collect(*palabra_clave*) **as** *conjunto_palabras_clave*
from *libros_plana*
group by *título, editor*

Otro enfoque de la creación de relaciones anidadas es usar subconsultas en la cláusula **select**. Una ventaja del enfoque de las subconsultas es que se puede usar de manera opcional en la subconsulta una cláusula **order by** para generar los resultados en el orden deseado, lo que puede aprovecharse para crear un array. La siguiente consulta ilustra este enfoque; las palabras clave **array** y **multiset** especifican que se van a crear un array y un multiconjunto (respectivamente) a partir del resultado de las subconsultas.

título	*autor*	*editor*	*conjunto_palabras_clave*
		(*nombre_editor*, *sucursal_editor*)	
Compiladores	Gómez	(McGraw-Hill, Nueva York)	{análisis sintáctico, análisis}
Compiladores	Santos	(McGraw-Hill, Nueva York)	{análisis sintáctico, análisis}
Redes	Santos	(Oxford, Londres)	{Internet, Web}
Redes	Escudero	(Oxford, Londres)	{Internet, Web}

Figura 9.4 Una versión parcialmente anidada de la relación *libros_plana*.

```
select título,
    array( select autores
      from autores as A
      where A.título = L.título
      order by A.posición) as array_autores,
    Editor(nombre_editor, sucursal_editor) as editor,
    multiset( select palabra_clave
      from palabras_clave as P
      where P.título = L.título) as conjunto_palabras_clave,
from libros4 as L
```

El sistema ejecuta las subconsultas anidadas de la cláusula **select** para cada tupla generada por las cláusulas **from** and *where* de la consulta externa. Obsérvese que el atributo *L.título* de la consulta externa se usa en las consultas anidadas para garantizar que sólo se generen los conjuntos correctos de autores y de palabras clave para cada título.

SQL:2003 ofrece gran variedad de operadores para multiconjuntos, incluida la funcion **set**(M), que calcula una versión libre duplicada del multiconjunto M, la operación agregada **intersection**, cuyo resultado es la intersección de todos los multiconjuntos de un grupo, la operación agregada **fusion**, que devuelve la unión de todos los multiconjuntos de un grupo, y el predicado **submultiset**, que comprueba si el multiconjunto está contenido en otro multiconjunto.

La norma de SQL no proporciona ningún medio para actualizar los atributos de los multiconjuntos, salvo asignarles valores nuevos. Por ejemplo, para borrar el valor v del atributo de multiconjunto A hay que definirlo como (A **except all multiset**[v]).

9.6 Identidad de los objetos y tipos de referencia en SQL

Los lenguajes orientados a objetos ofrecen la posibilidad de hacer referencia a objetos. Los atributos de un tipo dado pueden servir de referencia para los objetos de un tipo concreto. Por ejemplo, en SQL se puede definir el tipo *Departamento* con el campo *nombre* y el campo *director*, que es una referencia al tipo *Persona*, y la tabla *departamentos* del tipo *Departamento*, de la manera siguiente:

```
create type Departmento (
      nombre varchar(20),
      director ref(Persona) scope personas
)
create table departamentos of Departmento
```

En este caso, la referencia está restringida a las tuplas de la tabla *personas*. La restricción del **ámbito** (scope) de referencia a las tuplas de una tabla es obligatoria en SQL, y hace que las referencias se comporten como las claves externas.

Se puede omitir la declaración **scope** *personas* de la declaración de tipos y hacer, en su lugar, un añadido a la instrucción **create table**:

```
create table departamentos of Departmento
      (director with options scope personas)
```

La tabla a la que se hace referencia debe tener un atributo que guarde el identificador de cada tupla. Ese atributo, denominado **atributo autorreferencial** (self-referential attribute), se declara añadiendo una cláusula **ref is** a la instrucción **create table**:

```
create table personas of Persona
ref is id_personal system generated
```

En este caso, *id_personal* es el nombre de un atributo, no una palabra clave, y la instrucción **create table** especifica que la base de datos genera de manera automática el identificador.

Para inicializar el atributo de referencia hay que obtener el identificador de la tupla a la que se va a hacer referencia. Se puede conseguir el valor del identificador de la tupla mediante una consulta. Por tanto, para crear una tupla con el valor de referencia, primero se puede crear la tupla con una referencia nula y luego definir la referencia de manera independiente:

```
insert into departamentos
    values ('CS', null)
update departamentos
    set director = (select p.id_personal
                    from personal as p
                    where nombre = 'Martín')
where nombre = 'CS'
```

Una alternativa a los identificadores generados por el sistema es permitir que los usuarios generen los identificadores. El tipo del atributo autorreferencial debe especificarse como parte de la definición de tipos de la tabla a la que se hace referencia, y la definición de la tabla debe especificar que la referencia está **generada por el usuario** (**user generated**):

```
create type Persona
    (nombre varchar(20),
     dirección varchar(20))
    ref using varchar(20)
create table personas of Persona
    ref is id_personal user generated
```

Al insertar tuplas en *personas* hay que proporcionar el valor del identificador:

```
insert into personas (id_personal, nombre, dirección) values
    ('01284567', 'Martín', 'Avenida del Segura, 23')
```

Ninguna otra tupla de *personas*, de sus supertablas ni de sus subtablas puede tener el mismo identificador. Por tanto, se puede usar el valor del identificador al insertar tuplas en *departamentos*, sin necesidad de más consultas para recuperarlo:

```
insert into departamentos
    values ('CS', '01284567')
```

Incluso es posible usar el valor de una clave primaria ya existente como identificador, incluyendo la cláusula **ref from** en la definición de tipos:

```
create type Persona
    (nombre varchar(20) primary key,
     dirección varchar(20))
    ref from(nombre)
create table personas of Persona
    ref is id_personal derived
```

Téngase en cuenta que la definición de la tabla debe especificar que las referencia es derivada, y debe seguir especificando el nombre de un atributo autorreferencial. Al insertar una tupla de *departamentos* se puede usar

```
insert into departamentos
    values ('CS', 'Martín')
```

En SQL:1999 las referencias se desvinculan mediante el símbolo $->$. Considérese la tabla *departamentos* que se ha definido anteriormente. Se puede usar esta consulta para averiguar el nombre y la dirección de los directores de todos los departamentos:

select *director−>nombre*, *director−>dirección*
from *departamentos*

Las expresiones como "*director−>nombre*" se denominan **expresiones de camino**.

Dado que *director* es una referencia a una tupla de la tabla *personas*, el atributo *nombre* de la consulta anterior es el atributo *nombre* de la tupla de la tabla *personas*. Se pueden usar las referencias para ocultar las operaciones de reunión; en el ejemplo anterior, sin las referencias, el campo *director* de *departamento* se declararía como clave externa de la tabla *personas*. Para averiguar el nombre y la dirección del director de un departamento, hace falta una reunión explícita de las relaciones *departamentos* y *personas*. El uso de referencias simplifica considerablemente la consulta.

Se puede usar la operación **deref** para devolver la tupla a la que señala una referencia y luego tener acceso a sus atributos, como se muestra a continuación.

select deref(*director*).*nombre*
from *departamentos*

9.7 Implementación de las características O-R

Los sistemas de bases de datos relacionales orientadas a objetos son básicamente extensiones de los sistemas de bases de datos relacionales ya existentes. Las modificaciones resultan claramente necesarias en muchos niveles del sistema de bases de datos. Sin embargo, para minimizar las modificaciones en el código del sistema de almacenamiento (almacenamiento de relaciones, índices, etc.), los tipos de datos complejos soportados por los sistemas relacionales orientados a objetos se pueden traducir al sistema de tipos más sencillo de las bases de datos relacionales.

Para comprender la manera de hacer esta traducción, sólo hace falta mirar al modo en que algunas características del modelo E-R se traducen en relaciones. Por ejemplo, los atributos multivalorados del modelo E-R se corresponden con los atributos valorados como multiconjuntos del modelo relacional orientado a objetos. Los atributos compuestos se corresponden grosso modo con los tipos estructurados. Las jerarquías ES del modelo E-R se corresponden con la herencia de tablas del modelo relacional orientado a objetos.

Las técnicas para convertir las características del modelo E-R en tablas, que se estudiaron en el Apartado 6.9, se pueden usar, con algunas extensiones, para traducir los datos relacionales orientados a objetos a datos relacionales en el nivel de almacenamiento.

Las subtablas se pueden almacenar de manera eficiente, sin réplica de todos los campos heredados, de una de estas maneras:

- Cada tabla almacena la clave primaria (que puede haber heredado de una tabla madre) y los atributos que se definen de localmente. No hace falta almacenar los atributos heredados (que no sean la clave primaria), se pueden obtener mediante una reunión con la supertabla, de acuerdo con la clave primaria.
- Cada tabla almacena todos los atributos heredados y definidos localmente. Cuando se inserta una tupla, sólo se almacena en la tabla en la que se inserta, y su presencia se infiere en cada una de las supertablas. El acceso a todos los atributos de las tuplas es más rápido, ya que no hace falta ninguna reunión.

  No obstante, en caso de que el sistema de tipos permita que cada entidad se represente en dos subtablas sin estar presente en una subtabla común a ambas, esta representación puede dar lugar a la réplica de información. Además, resulta difícil traducir las claves externas que hacen referencia a una supertabla en restricciones de las subtablas; para implementar de manera eficiente esas claves externas hay que definir la supertabla como vista, y el sistema de bases de datos tiene que soportar las claves externas en las vistas.

Las implementaciones pueden decidir representar los tipos arrays y multiconjuntos directamente o usar internamente una representación normalizada. Las representaciones normalizadas tienden a ocupar más espacio y exigen un coste adicional en reuniones o agrupamientos para reunir los datos en

arrays o en multiconjuntos. Sin embargo, puede que las representaciones normalizadas resulten más sencillas de implementar.

Las interfaces de programas de aplicación ODBC y JDBC se han extendido para recuperar y almacenar tipos estructurados; por ejemplo, JDBC ofrece el método getObject(), que es parecido a getString() pero devuelve un objeto Java Struct, a partir del cual se pueden extraer los componentes del tipo estructurado. También es posible asociar clases de Java con tipos estructurados de SQL, y JDBC puede realizar la conversión entre los tipos. Véase el manual de referencia de ODBC o de JDBC para obtener más detalles.

9.8 Lenguajes de programación persistentes

Los lenguajes de las bases de datos se diferencian de los lenguajes de programación tradicionales en que trabajan directamente con datos que son persistentes; es decir, los datos siguen existiendo una vez que el programa que los creó haya concluido. Las relaciones de las bases de datos y las tuplas de las relaciones son ejemplos de datos persistentes. Por el contrario, los únicos datos persistentes con los que los lenguajes de programación tradicionales trabajan directamente son los archivos.

El acceso a las bases de datos es sólo un componente de las aplicaciones del mundo real. Mientras que los lenguajes para el tratamiento de datos como SQL son bastante efectivos en el acceso a los datos, se necesita un lenguaje de programación para implementar otros componentes de las aplicaciones como las interfaces de usuario o la comunicación con otras computadoras. La manera tradicional de realizar las interfaces de las bases de datos con los lenguajes de programación es incorporar SQL dentro del lenguaje de programación.

Los **lenguajes de programación persistentes** son lenguajes de programación extendidos con estructuras para el tratamiento de los datos persistentes. Los lenguajes de programación persistentes pueden distinguirse de los lenguajes con SQL incorporado, al menos, de dos maneras:

1. En los lenguajes incorporados el sistema de tipos del lenguaje anfitrión suele ser diferente del sistema de tipos del lenguaje para el tratamiento de los datos. Los programadores son responsables de las conversiones de tipos entre el lenguaje anfitrión y SQL. Hacer que los programadores lleven a cabo esta tarea presenta varios inconvenientes:
   - El código para la conversión entre objetos y tuplas opera fuera del sistema de tipos orientado a objetos y, por tanto, tiene más posibilidades de presentar errores no detectados.
   - La conversión en la base de datos entre el formato orientado a objetos y el formato relacional de las tuplas necesita gran cantidad de código. El código para la conversión de formatos, junto con el código para cargar y descargar los datos de la base de datos, puede suponer un porcentaje significativo del código total necesario para la aplicación.

   Por el contrario, en los lenguajes de programación persistentes, el lenguaje de consultas se halla totalmente integrado con el lenguaje anfitrión y ambos comparten el mismo sistema de tipos. Los objetos se pueden crear y guardar en la base de datos sin ninguna modificación explícita del tipo o del formato; los cambios de formato necesarios se realizan de manera transparente.
2. Los programadores que usan lenguajes de consultas incorporados son responsables de la escritura de código explícito para la búsqueda en la memoria de los datos de la base de datos. Si se realizan actualizaciones, los programadores deben escribir explícitamente código para volver a guardar los datos actualizados en la base de datos.

   Por el contrario, en los lenguajes de programación persistentes, los programadores pueden trabajar con datos persistentes sin tener que escribir explícitamente código para buscarlos en la memoria o volver a guardarlos en el disco.

En este apartado se describe la manera en que se pueden extender los lenguajes de programación orientados a objetos, como C++ y Java, para hacerlos lenguajes de programación persistentes. Las características de estos lenguajes permiten que los programadores trabajen con los datos directamente desde el lenguaje de programación, sin tener que recurrir a lenguajes de tratamiento de datos como SQL. Por tanto, ofrecen una integración más estrecha de los lenguajes de programación con las bases de datos que, por ejemplo, SQL incorporado.

Sin embargo, los lenguajes de programación persistentes presentan ciertos inconvenientes que hay que tener presentes al decidir si conviene usarlos. Dado que los lenguajes de programación suelen ser potentes, resulta relativamente sencillo cometer errores de programación que dañen las bases de datos. La complejidad de los lenguajes hace que la optimización automática de alto nivel, como la reducción de E/S de disco, resulte más difícil. En muchas aplicaciones el soporte de las consultas declarativas resulta de gran importancia, pero los lenguajes de programación persistentes no soportan bien actualmente las consultas declarativas.

En este capítulo se describen varios problemas teóricos que hay que abordar a la hora de añadir la persistencia a los lenguajes de programación ya existentes. En primer lugar se abordan los problemas independientes de los lenguajes y, en apartados posteriores, se tratan problemas que son específicos de los lenguajes C++ y Java. No obstante, no se tratan los detalles de las extensiones de los lenguajes; aunque se han propuesto varias normas, ninguna ha tenido una aceptación universal. Véanse las referencias de las notas bibliográficas para saber más sobre extensiones de lenguajes concretas y más detalles de sus implementaciones.

9.8.1 Persistencia de los objetos

Los lenguajes de programación orientados a objetos ya poseen un concepto de objeto, un sistema de tipos para definir los tipos de los objetos y constructores para crearlos. Sin embargo, esos objetos son *transitorios*; desaparecen en cuanto finaliza el programa, igual que ocurre con las variables de los programas en Pascal o en C. Si se desea transformar uno de estos lenguajes en un lenguaje para la programación de bases de datos, el primer paso consiste en proporcionar una manera de hacer persistentes a los objetos. Se han propuesto varios enfoques.

- **Persistencia por clases**. El enfoque más sencillo, pero el menos conveniente, consiste en declarar que una clase es persistente. Todos los objetos de la clase son, por tanto, persistentes de manera predeterminada. Todos los objetos de las clases no persistentes son transitorios.

  Este enfoque no es flexible, dado que suele resultar útil disponer en una misma clase tanto de objetos transitorios como de objetos persistentes. Muchos sistemas de bases de datos orientados a objetos interpretan la declaración de que una clase es persistente como si se afirmara que los objetos de la clase pueden hacerse persistentes, en vez de que todos los objetos de la clase son persistentes. Estas clases se pueden denominar con más propiedad clases "que pueden ser persistentes".

- **Persistencia por creación**. En este enfoque se introduce una sintaxis nueva para crear los objetos persistentes mediante la extensión de la sintaxis para la creación de los objetos transitorios. Por tanto, los objetos son persistentes o transitorios en función de la forma de crearlos. Varios sistemas de bases de datos orientados a objetos siguen este enfoque.

- **Persistencia por marcas**. Una variante del enfoque anterior es marcar los objetos como persistentes después de haberlos creado. Todos los objetos se crean como transitorios pero, si un objeto tiene que persistir más allá de la ejecución del programa, hay que marcarlo como persistente de manera explícita antes de que éste concluya. Este enfoque, a diferencia del anterior, pospone la decisión sobre la persistencia o la transitoriedad hasta después de la creación del objeto.

- **Persistencia por alcance**. Uno o varios objetos se declaran objetos persistentes (objetos raíz) de manera explícita. Todos los demás objetos serán persistentes si (y sólo si) se pueden alcanzar desde algún objeto raíz mediante una secuencia de una o varias referencias.

  Por tanto, todos los objetos a los que se haga referencia desde (es decir, cuyos identificadores de objetos se guarden en) los objetos persistentes raíz serán persistentes. Pero también lo serán todos los objetos a los que se haga referencia desde ellos, y los objetos a los que éstos últimos hagan referencia serán también persistentes, etc.

  Una ventaja de este esquema es que resulta sencillo hacer que sean persistentes estructuras de datos completas con sólo declarar como persistente su raíz. Sin embargo, el sistema de bases de datos sufre la carga de tener que seguir las cadenas de referencias para detectar los objetos que son persistentes, y eso puede resultar costoso.

9.8.2 Identidad de los objetos y punteros

En los lenguajes de programación orientados a objetos que no se han extendido para tratar la persistencia, cuando se crea un objeto el sistema devuelve un identificador del objeto transitorio. Los identificadores de objetos transitorios sólo son válidos mientras se ejecuta el programa que los ha creado; después de que concluya ese programa, el objeto se borra y el identificador pierde su sentido. Cuando se crea un objeto persistente se le asigna un identificador de objeto persistente.

El concepto de identidad de los objetos tiene una relación interesante con los punteros de los lenguajes de programación. Una manera sencilla de conseguir una identidad intrínseca es usar los punteros a las ubicaciones físicas de almacenamiento. En concreto, en muchos lenguajes orientados a objetos como C++, los identificadores de los objetos son en realidad punteros internos de la memoria.

Sin embargo, la asociación de los objetos con ubicaciones físicas de almacenamiento puede variar con el tiempo. Hay varios grados de permanencia de las identidades:

- **Dentro de los procedimientos**. La identidad sólo persiste durante la ejecución de un único procedimiento. Un ejemplo de identidad dentro de los programas son las variables locales de los procedimientos.
- **Dentro de los programas**. La identidad sólo persiste durante la ejecución de un único programa o de una única consulta. Un ejemplo de identidad dentro de los programas son las variables globales de los lenguajes de programación. Los punteros de la memoria principal o de la memoria virtual sólo ofrecen identidad dentro de los programas.
- **Entre programas**. La identidad persiste de una ejecución del programa a otra. Los punteros a los datos del sistema de archivos del disco ofrecen identidad entre los programas, pero pueden cambiar si se modifica la manera en que los datos se guardan en el sistema de archivos.
- **Persistente**. La identidad no sólo persiste entre las ejecuciones del programa sino también entre reorganizaciones estructurales de los datos. Es la forma persistente de identidad necesaria para los sistemas orientados a objetos.

En las extensiones persistentes de los lenguajes como C++, los identificadores de objetos de los objetos persistentes se implementan como "punteros persistentes". Un *puntero persistente* es un tipo de puntero que, a diferencia de los punteros internos de la memoria, sigue siendo válido después del final de la ejecución del programa y después de algunas modalidades de reorganización de los datos. Los programadores pueden usar los punteros persistentes del mismo modo que usan los punteros internos de la memoria en los lenguajes de programación. Conceptualmente los punteros persistentes se pueden considerar como punteros a objetos de la base de datos.

9.8.3 Almacenamiento y acceso a los objetos persistentes

¿Qué significa guardar un objeto en una base de datos? Evidentemente, hay que guardar por separado la parte de datos de cada objeto. Lógicamente, el código que implementa los métodos de las clases debe guardarse en la base de datos como parte de su esquema, junto con las definiciones de tipos de las clases. Sin embargo, muchas implementaciones se limitan a guardar el código en archivos externos a la base de datos para evitar tener que integrar el software del sistema, como los compiladores, con el sistema de bases de datos.

Hay varias maneras de hallar los objetos de la base de datos. Una manera es dar nombres a los objetos, igual que se hace con los archivos. Este enfoque funciona con un número de objetos relativamente pequeño, pero no resulta práctico para millones de objetos. Una segunda manera es exponer los identificadores de los objetos o los punteros persistentes a los objetos, que pueden guardarse en el exterior. A diferencia de los nombres, los punteros no tienen por qué ser fáciles de recordar y pueden ser, incluso, punteros físicos internos de la base de datos.

Una tercera manera es guardar conjuntos de objetos y permitir que los programas iteren sobre ellos para buscar los objetos deseados. Los conjuntos de objetos pueden a su vez modelarse como objetos de un *tipo conjunto*. Entre los tipos de conjuntos están los conjuntos, los multiconjuntos (es decir, conjuntos con varias apariciones posibles de un mismo valor), las listas, etc. Un caso especial de conjunto son

las *extensiones de clases*, que son el conjunto de todos los objetos pertenecientes a una clase. Si hay una extensión de clase para una clase dada, siempre que se crea un objeto de la clase ese objeto se inserta en la extensión de clase de manera automática; y, siempre que se borra un objeto, éste se elimina de la extensión de clase. Las extensiones de clases permiten que las clases se traten como relaciones en el sentido de que es posible examinar todos los objetos de una clase, igual que se pueden examinar todas las tuplas de una relación.

La mayor parte de los sistemas de bases de datos orientados a objetos soportan las tres maneras de acceso a los objetos persistentes. Dan identificadores a todos los objetos. Generalmente sólo dan nombre a las extensiones de las clases y a otros objetos de tipo conjunto y, quizás, a otros objetos seleccionados, pero no a la mayor parte de los objetos. Las extensiones de las clases suelen conservarse para todas las clases que puedan tener objetos persistentes pero, en muchas de las implementaciones, las extensiones de las clases sólo contienen los objetos persistentes de cada clase.

9.8.4 Sistemas persistentes de C++

En los últimos años han aparecido varias bases de datos orientadas a objetos basadas en las extensiones persistentes de C++ (véanse las notas bibliográficas). Hay diferencias entre ellas en términos de la arquitectura de los sistemas pero tienen muchas características comunes en términos del lenguaje de programación.

Varias de las características orientadas a objetos del lenguaje C++ ayudan a proporcionar un buen soporte para la persistencia sin modificar el propio lenguaje. Por ejemplo, se puede declarar una clase denominada Persistent_Object (objeto persistente) con los atributos y los métodos para dar soporte la persistencia; cualquier otra clase que deba ser persistente puede hacerse subclase de esta clase y heredará, por tanto, el soporte de la persistencia. El lenguaje C++ (igual que otros lenguajes modernos de programación) permite también redefinir los nombres de las funciones y los operadores estándar—como $+$, $-$, el operador de desvinculación de los punteros $->$, etc.—en función del tipo de operandos a los que se aplican. Esta posibilidad se denomina *sobrecarga*; se usa para redefinir los operadores para que se comporten de la manera deseada cuando operan con objetos persistentes.

Proporcionar apoyo a la persistencia mediante las bibliotecas de clases presenta la ventaja de que sólo se realizan cambios mínimos en C++; además, resulta relativamente fácil de implementar. Sin embargo, presenta el inconveniente de que los programadores tienen que usar mucho más tiempo para escribir los programas que trabajan con objetos persistentes y de que no les resulta sencillo especificar las restricciones de integridad del esquema ni ofrecer soporte para las consultas declarativas. Algunas implementaciones persistentes de C++ soportan extensiones de la sintaxis de C++ para facilitar estas tareas.

Es necesario abordar los siguientes problemas a la hora de añadir soporte a la persistencia a C++ (y a otros lenguajes):

- **Punteros persistentes**. Se debe definir un nuevo tipo de datos para que represente los punteros persistentes. Por ejemplo, la norma ODMG de C++ define la clase de plantillas d_Ref$< T >$ para que represente los punteros persistentes a la clase T. El operador de desvinculación para esta clase se vuelve a definir para que capture el objeto del disco (si no se halla ya presente en la memoria) y devuelva un puntero de la memoria al búfer en el que se ha capturado el objeto. Por tanto, si p es un puntero persistente a la clase T, se puede usar la sintaxis estándar como p$->$A o p$->$f(v) para tener acceso al atributo A de la clase T o para invocar al método f de la clase T.

  El sistema de bases de datos ObjectStore usa un enfoque diferente de los punteros persistentes. Utiliza los tipos normales de punteros para almacenar los punteros persistentes. Esto plantea dos problemas: (1) el tamaño de los punteros de la memoria sólo puede ser de 4 bytes, demasiado pequeño para las bases de datos mayores de 4 gigabytes, y (2) cuando se pasa algún objeto al disco los punteros de la memoria que señalan a su antigua ubicación física carecen de significado. ObjectStore usa una técnica denominada "rescate hardware" para abordar ambos problemas; precaptura los objetos de la base de datos en la memoria y sustituye los punteros persistentes por punteros de memoria y, cuando se vuelven a almacenar los datos en el disco, los punteros de memoria se sustituyen por punteros persistentes. Cuando se hallan en el disco, el valor almacenado

en el campo de los punteros de memoria no es el puntero persistente real; en vez de eso, se busca el valor en una tabla para averiguar el valor completo del puntero persistente.

- **Creación de objetos persistentes**. El operador new de C++ se usa para crear objetos persistentes mediante la definición de una versión "sobrecargada" del operador que usa argumentos adicionales especificando que deben crearse en la base de datos. Por tanto, en lugar de new T(), hay que llamar a new (bd) T() para crear un objeto persistente, donde bd identifica a la base de datos.
- **Extensiones de las clases**. Las extensiones de las clases se crean y se mantienen de manera automática para cada clase. La norma ODMG de C++ exige que el nombre de la clase se pase como parámetro adicional a la operación new. Esto también permite mantener varias extensiones para cada clase, pasando diferentes nombres.
- **Relaciones**. Las relaciones entre las clases se suelen representar almacenando punteros de cada objeto a los objetos con los que está relacionado. Los objetos relacionados con varios objetos de una clase dada almacenan un conjunto de punteros. Por tanto, si un par de objetos se halla en una relación, cada uno de ellos debe almacenar un puntero al otro. Los sistemas persistentes de C++ ofrecen una manera de especificar esas restricciones de integridad y de hacer que se cumplan mediante la creación y borrado automático de los punteros. Por ejemplo, si se crea un puntero de un objeto a a un objeto b, se añade de manera automática un puntero a a al objeto b.
- **Interfaz iteradora**. Dado que los programas tienen que iterar sobre los miembros de las clases, hace falta una interfaz para iterar sobre los miembros de las extensiones de las clases. La interfaz iteradora también permite especificar selecciones, de modo que sólo hay que capturar los objetos que satisfagan el predicado de selección.
- **Transacciones**. Los sistemas persistentes de C++ ofrecen soporte para comenzar las transacciones y para comprometerlas o provocar su retroceso.
- **Actualizaciones**. Uno de los objetivos de ofrecer soporte a la persistencia a los lenguajes de programación es permitir la persistencia transparente. Es decir, una función que opere sobre un objeto no debe necesitar saber que el objeto es persistente; por tanto, se pueden usar las mismas funciones sobre los objetos independientemente de que sean persistentes o no.

  Sin embargo, surge el problema de que resulta difícil detectar cuándo se ha actualizado un objeto. Algunas extensiones persistentes de C++ exigían que el programador especificara de manera explícita que se ha modificado el objeto llamando a la función mark_modified(). Además de incrementar el esfuerzo del programador, este enfoque aumenta la posibilidad de que se produzcan errores de programación que den lugar a una base de datos corrupta. Si un programador omite la llamada a mark_modified(), es posible que nunca se propague a la base de datos la actualización llevada a cabo por una transacción, mientras que otra actualización realizada por la misma transacción sí se propaga, lo que viola la atomicidad de las transacciones.

  Otros sistemas, como ObjectStore, usan soporte a la protección de la memoria proporcionada por el sistema operativo o por el hardware para detectar las operaciones de escritura en los bloques de memoria y marcar como sucios los bloques que se deban escribir posteriormente en el disco.
- **Lenguaje de consultas**. Los iteradores ofrecen soporte para consultas de selección sencillas. Para soportar consultas más complejas los sistemas persistentes de C++ definen un lenguaje de consultas.

Gran número de sistemas de bases de datos orientados a objetos basados en C++ se desarrollaron a finales de los años ochenta y principios de los noventa del siglo veinte. Sin embargo, el mercado para esas bases de datos resultó mucho más pequeño de lo esperado, ya que la mayor parte de los requisitos de las aplicaciones se cumplen de sobra usando SQL mediante interfaces como ODBC o JDBC. En consecuencia, la mayor parte de los sistemas de bases de datos orientados a objetos desarrollados en ese periodo ya no existen. En los años noventa el Grupo de Gestión de Datos de Objetos (Object Data Management Group, ODMG) definió las normas para agregar persistencia a C++ y a Java. No obstante,

el grupo concluyó sus actividades alrededor de 2002. ObjectStore y Versant son de los pocos sistemas de bases de datos orientados a objetos originales que siguen existiendo.

Aunque los sistemas de bases de datos orientados a objetos no encontraron el éxito comercial que esperaban, la razón de añadir persistencia a los lenguajes de programación sigue siendo válida. Hay varias aplicaciones con grandes exigencias de rendimiento que se ejecutan en sistemas de bases de datos orientados a objetos; el uso de SQL impondría una sobrecarga de rendimiento excesiva para muchos de esos sistemas. Con los sistemas de bases de datos relacionales orientados a objetos que proporcionan soporte para los tipos de datos complejos, incluidas las referencias, resulta más sencillo almacenar los objetos de los lenguajes de programación en bases de datos de SQL. Todavía puede emerger una nueva generación de sistemas de bases de datos orientadas a objetos que utilicen bases de datos relacionales orientadas a objetos como sustrato.

9.8.5 Sistemas Java persistentes

En años recientes el lenguaje Java ha visto un enorme crecimiento en su uso. La demanda de soporte de la persistencia de los datos en los programas de Java se ha incrementado de manera acorde. Los primeros intentos de creación de una norma para la persistencia en Java fueron liderados por el consorcio ODMG; posteriormente, el consorcio concluyó sus esfuerzos, pero transfirió su diseño al proyecto **Objetos de bases de datos de Java** (Java Database Objects, JDO), que coordina Sun Microsystems.

El modelo JDO para la persistencia de los objetos en los programas de Java es diferente del modelo de soporte de la persistencia en los programas de C++. Entre sus características se hallan:

- **Persistencia por alcance**. Los objetos no se crean explícitamente en la base de datos. El registro explícito de un objeto como persistente (usando el método makePersistent() de la clase PersistenceManager) hace que el objeto sea persistente. Además, cualquier objeto alcanzable desde un objeto persistente pasa a ser persistente.
- **Mejora del código de bytes**. En lugar de declarar en el código de Java que una clase es persistente, se especifican en un archivo de configuración (con la extensión .jdo) las clases cuyos objetos se pueden hacer persistentes. Se ejecuta un programa *mejorador* específico de la implementación que lee el archivo de configuración y lleva a cabo dos tareas. En primer lugar, puede crear estructuras en la base de datos para almacenar objetos de esa clase. En segundo lugar, modifica el código de bytes (generado al compilar el programa de Java) para que maneje tareas relacionadas con la persistencia. A continuación se ofrecen algunos ejemplos de modificaciones de este tipo.
  - Se puede modificar cualquier código que tenga acceso a un objeto para que compruebe primero si el objeto se halla en la memoria y, si no está, dé los pasos necesarios para ponerlo en la memoria.
  - Cualquier código que modifique un objeto se modifica para que también registre el objeto como modificado y, quizás, para que guarde un valor previo a la actualización que se usa en caso de que haga falta deshacerla (es decir, si se retrocede la transacción).

  También se pueden llevar a cabo otras modificaciones del código de bytes. Esas modificaciones del código de bytes son posibles porque el código de bytes es estándar en todas las plataformas e incluye mucha más información que el código objeto compilado.
- **Asignación de bases de datos**. JDO no define la manera en que se almacenan los datos en la base de datos subyacente. Por ejemplo, una situación frecuente es que los objetos se almacenen en una base de datos relacional. El programa mejorador puede crear en la base de datos un esquema adecuado para almacenar los objetos de las clases. La manera exacta en que lo hace depende de la implementación y no está definida por JDO. Se pueden asignar algunos atributos a los atributos relacionales, mientras que otros se pueden almacenar de forma serializada, que la base de datos trata como si fuera un objeto binario. Las implementaciones de JDO pueden permitir que los datos relacionales existentes se vean como objetos mediante la definición de la asignación correspondiente.
- **Extensiones de clase**. Las extensiones de clase se crean y se conservan de manera automática para cada clase declarada como persistente. Todos los objetos que se hacen persistentes se añaden

de manera automática a la extensión de clase correspondiente a a su clase. Los programas de JDO pueden tener acceso a las extensiones de clase e iterar sobre los miembros seleccionados. La interfaz Iteradora ofrecida por Java se puede usar para crear iteradores sobre las extensiones de clase y avanzar por los miembros de cada extensión de clase. JDO también permite que se especifiquen selecciones cuando se crea una extensión de clase y que sólo se capturen los objetos que satisfagan la selección.

- **Tipo de referencia único**. No hay diferencia de tipos entre las referencias a los objetos transitorios y las referencias a los objetos persistentes.

  Un enfoque para conseguir esta unificación de los tipos de puntero es cargar toda la base de datos en la memoria, lo que sustituye todos los punteros persistentes por punteros de memoria. Una vez llevadas a cabo las actualizaciones, el proceso se invierte y se vuelven a almacenar en el disco los objetos actualizados. Este enfoque es muy ineficiente para bases de datos de gran tamaño.

  A continuación se describirá un enfoque alternativo, que permite que los objetos persistentes se capturen en memoria de manera automática cuando hace falta, mientras que permite que todas las referencias contenidas en objetos de la memoria sean referencias de la memoria. Cuando se captura el objeto A, se crea un **objeto hueco** para cada objeto B_i al que haga referencia, y la copia de la memoria de A tiene referencias al objeto hueco correspondiente para cada B_i. Por supuesto, el sistema tiene que asegurar que, si ya se ha capturado el objeto B_i, la referencia apunta al objeto ya capturado en lugar de crear un nuevo objeto hueco. De manera parecida, si no se ha capturado el objeto B_i, pero otro objeto ya capturado hace referencia a él, ya tiene un objeto hueco creado para él; la referencia al objeto hueco ya existente se vuelve a usar, en lugar de crear un objeto hueco nuevo.

  Por tanto, para cada objeto O_i que se haya capturado, cada referencia de O_i es a un objeto ya capturado o a un objeto hueco. Los objetos huecos forman un *borde* que rodea los objetos capturados.

  Siempre que el programa tiene acceso real al objeto hueco O, el código de bytes mejorado lo detecta y captura el objeto en la base de datos. Cuando se captura este objeto, se lleva a cabo el mismo proceso de creación de objetos huecos para todos los objetos a los que O hace referencia. Tras esto, se permite que se lleve a cabo el acceso al objeto[3].

  Para implementar este esquema hace falta una estructura de índices en la memoria que asigne los punteros persistentes a las referencias de la memoria. Cuando se vuelven a escribir los objetos en el disco se usa ese índice para sustituir las referencias de la memoria por punteros persistentes en la copia que se escribe en el disco.

La norma JDO se halla todavía en una etapa preliminar y sometida a revisión. Varias empresas ofrecen implementaciones de JDO. No obstante, queda por ver si JDO se usará mucho, a diferencia de C++ de ODMG.

9.9 Sistemas orientados a objetos y sistemas relacionales orientados a objetos

Ya se han estudiado las bases de datos relacionales orientadas a objetos, que son bases de datos orientadas a objetos construidas sobre el modelo relacional, así como las bases de datos orientadas a objetos, que se crean alrededor de los lenguajes de programación persistentes.

3. La técnica que usa objetos huecos descrita anteriormente se halla estrechamente relacionada con la técnica de rescate hardware (ya mencionada en el Apartado 9.8.4). El rescate hardware la usan algunas implementaciones persistentes de C++ para ofrecer un solo tipo de puntero para los punteros persistentes y los de memoria. El rescate hardware utiliza técnicas de protección de la memoria virtual proporcionadas por el sistema operativo para detectar el acceso a las páginas y captura las páginas de la base de datos cuando es necesario. Por el contrario, la versión de Java modifica el código de bytes para que compruebe la existencia de objetos huecos en lugar de usar la protección de la memoria y captura los objetos cuando hace falta, en vez de capturar páginas enteras de la base de datos.

Las extensiones persistentes de loslenguajes de programación y los sistemas relacionales orientados a objetos se dirigen a mercados diferentes. La naturaleza declarativa y la limitada potencia (comparada con la de los lenguajes de programación) del lenguaje SQL proporcionan una buena protección de los datos respecto de los errores de programación y hacen que las optimizaciones de alto nivel, como la reducción de E/S, resulten relativamente sencillas (la optimización de las expresiones relacionales se trata en el Capítulo 14). Los sistemas relacionales orientados a objetos se dirigen a simplificar la realización de los modelos de datos y de las consultas mediante el uso de tipos de datos complejos. Entre las aplicaciones habituales están el almacenamiento y la consulta de datos complejos, incluidos los datos multimedia.

Los lenguajes declarativos como SQL, sin embargo, imponen una reducción significativa del rendimiento a ciertos tipos de aplicaciones que se ejecutan principalmente en la memoria principal y realizan gran número de accesos a la base de datos. Los lenguajes de programación persistentes se dirigen a las aplicaciones de este tipo que tienen necesidad de un rendimiento elevado. Proporcionan acceso a los datos persistentes con poca sobrecarga y eliminan la necesidad de traducir los datos si hay que tratarlos con un lenguaje de programación. Sin embargo, son más susceptibles de deteriorar los datos debido a los errores de programación y no suelen disponer de gran capacidad de consulta. Entre las aplicaciones habituales están las bases de datos de CAD.

Los puntos fuertes de los diversos tipos de sistemas de bases de datos pueden resumirse de la manera siguiente:

- **Sistemas relacionales**: tipos de datos sencillos, lenguajes de consultas potentes, protección elevada.
- **Bases de datos orientadas a objetos basadas en lenguajes de programación persistentes**: tipos de datos complejos, integración con los lenguajes de programación, elevado rendimiento.
- **Sistemas relacionales orientados a objetos**: tipos de datos complejos, lenguajes de consultas potentes, protección elevada.

Estas descripciones son válidas en general, pero hay que tener en cuenta que algunos sistemas de bases de datos no respetan estas fronteras. Por ejemplo, algunos sistemas de bases de datos orientados a objetos construidos alrededor de lenguajes de programación persistentes se pueden implementar sobre sistemas de bases de datos relacionales o sobre sistemas de bases de datos relacionales orientados a objetos. Puede que estos sistemas proporcionen menor rendimiento que los sistemas de bases de datos orientados a objetos construidos directamente sobre los sistemas de almacenamiento, pero proporcionan parte de las garantías de protección más estrictas propias de los sistemas relacionales.

9.10 Resumen

- El modelo de datos relacional orientado a objetos extiende el modelo de datos relacional al proporcionar un sistema de tipos enriquecido que incluye los tipos de conjuntos y la orientación a objetos.
- Los tipos de conjuntos incluyen las relaciones anidadas, los conjuntos, los multiconjuntos y los arrays, y el modelo relacional orientado a objetos permite que los atributos de las tablas sean conjuntos.
- La orientación a objetos proporciona herencia con subtipos y subtablas, así como referencias a objetos (tuplas).
- La norma SQL:1999 extiende el lenguaje de definición de datos y de consultas con los nuevos tipos de datos y la orientación a objetos.
- Se han estudiado varias características del lenguaje de definición de datos extendido, así como del lenguaje de consultas y, en especial, da soporte a los atributos valorados como conjuntos, a la herencia y a las referencias a las tuplas. Estas extensiones intentan conservar los fundamentos relacionales—en particular, el acceso declarativo a los datos—a la vez que se extiende la potencia de modelado.

- Los sistemas de bases de datos relacionales orientados a objetos (es decir, los sistemas de bases de datos basados en el modelo relacional orientado a objetos) ofrecen un camino de migración cómodo para los usuarios de las bases de datos relacionales que desean usar las características orientadas a objetos.
- Las extensiones persistentes de C++ y Javan integran la persistencia de forma elegante y ortogonalmente a sus elementos de programación previos, por lo que resulta fácil de usar.
- La norma ODMG define las clases y otros constructores para la creación y acceso a los objetos persistentes desde C++, mientras que la norma JDO ofrece una funcionalidad equivalente para Java.
- Se han estudiado las diferencias entre los lenguajes de programación persistentes y los sistemas relacionales orientados a objetos, y se han mencionado criterios para escoger entre ellos.

Términos de repaso

- Relaciones anidadas.
- Modelo relacional anidado.
- Tipos complejos.
- Tipos de conjuntos.
- Tipos de objetos grandes.
- Conjuntos.
- Arrays.
- Multiconjuntos.
- Tipos estructurados.
- Métodos.
- Tipos fila.
- Funciones constructoras.
- Herencia:
  - ☐ Simple.
  - ☐ Múltiple.
- Herencia de tipos.
- Tipo más concreto.
- Herencia de tablas.
- Subtabla.
- Solapamiento de subtablas.
- Tipos de referencia.
- Ámbito de una referencia.
- Atributo autorreferencial.
- Expresiones de camino.
- Anidamiento y desanidamiento.
- Funciones y procedimientos en SQL.
- Lenguajes de programación persistentes.
- Persistencia por:
  - ☐ Clase.
  - ☐ Creación.
  - ☐ Marcado.
  - ☐ Alcance.
- Enlace C++ de ODMG.
- ObjectStore.
- JDO
  - ☐ Persistencia por alcance.
  - ☐ Raíces.
  - ☐ Objetos huecos.
  - ☐ Asignación relacional de objetos.

Ejercicios prácticos

9.1 Una compañía de alquiler de coches tiene una base de datos con todos los vehículos de su flota actual. Para todos los vehículos incluye el número de bastidor, el número de matrícula, el fabricante, el modelo, la fecha de adquisición y el color. Se incluyen datos especiales para algunos tipos de vehículos:

- Camiones: capacidad de carga.
- Coches deportivos: potencia, edad mínima del arrendatario.
- Furgonetas: número de plazas.
- Vehículos todoterreno: altura de los bajos, eje motor (tracción a dos ruedas o a las cuatro).

Constrúyase una definición del esquema de esta base de datos de acuerdo con SQL:1999. Utilícese la herencia donde resulte conveniente.

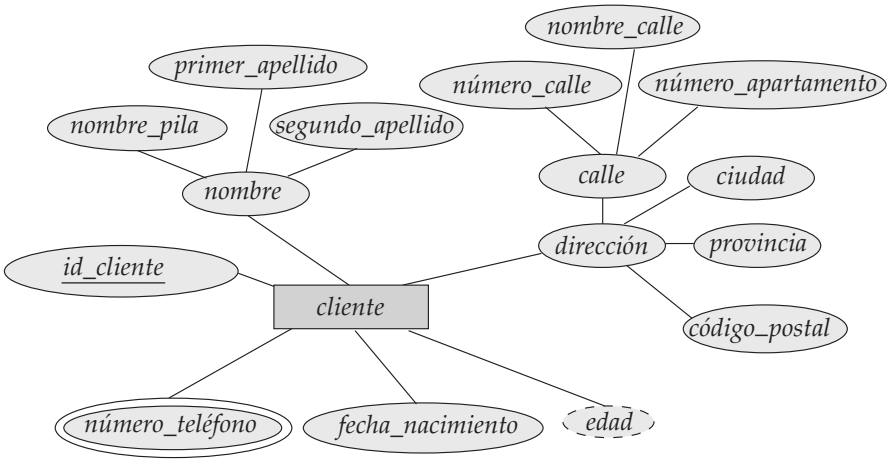

Figura 9.5 Diagrama E-R con atributos compuestos, multivalorados y derivados.

9.2 Considérese el esquema de la base de datos con la relación *Emp* cuyos atributos se muestran a continuación, con los tipos especificados para los atributos multivalorados.

Emp = (nombree, ConjuntoHijos **multiset***(HijosC), ConjuntoConocimientos* **multiset***(Conocimientos))*
Hijos = (nombre, cumpleaños)
Conocimientos = (mecanografía, ConjuntoExámenes **setof***(Exámenes))*
Exámenes = (año, ciudad)

a. Definir el esquema anterior en SQL:2003, con los tipos correspondientes para cada atributo.
b. Usando el esquema anterior, escribir las consultas siguientes en SQL:2003.
   I. Averiguar el nombre de todos los empleados que tienen un hijo nacido el 1 de enero de 2000 o en fechas posteriores.
   II. Averiguar los empleados que han hecho un examen del tipo de conocimiento "mecanografía" en la ciudad "San Rafael".
   III. Hacer una relación de los tipos de conocimiento de la relación *Emp*.

9.3 Considérese el diagrama E-R de la Figura 9.5, que contiene atributos compuestos, multivalorados y derivados.

a. Dese una definición de esquema en SQL:2003 correspondiente al diagrama E-R.
b. Dense constructores para cada uno de los tipos estructurados definidos.

9.4 Considérese el esquema relacional de la Figura 9.6.

a. Dese una definición de esquema en SQL:2003 correspondiente al esquema relacional, pero usando referencias para expresar las relaciones de clave externa.
b. Escríbanse cada una de las consultas del Ejercicio 2.9 sobre el esquema anterior usando SQL:2003.

9.5 Supóngase que se trabaja como asesor para escoger un sistema de bases de datos para la aplicación del cliente. Para cada una de las aplicaciones siguientes indíquese el tipo de sistema de bases de datos (relacional, base de datos orientada a objetos basada en un lenguaje de programación per-

empleado (*nombre_empleado*, *calle*, *ciudad*)
trabaja (*nombre_empleado*, *nombre_empresa*, sueldo)
empresa (*nombre_empresa*, *ciudad*)
supervisa (*nombre_empleado*, *nombre_jefe*)

Figura 9.6 Base de datos relacional para el Ejercicio práctico 9.4.

sistente, relacional orientada a objetos; no se debe especificar ningún producto comercial) que se recomendaría. Justifíquese la recomendación.

a. Sistema de diseño asistido por computadora para un fabricante de aviones.
b. Sistema para realizar el seguimiento de los donativos hechos a los candidatos a un cargo público
c. Sistema de información de ayuda para la realización de películas.

9.6 ¿En qué se diferencia el concepto de objeto del modelo orientado a objetos del concepto de entidad del modelo entidad-relación?

Ejercicios

9.7 Vuélvase a diseñar la base de datos del Ejercicio práctico 9.2 en la primera y en la cuarta formas normales. Indíquense las dependencias funcionales o multivaloradas que se den por supuestas. Indíquense también todas las restricciones de integridad referencial que deban incluirse en los esquemas de la primera y de la cuarta formas normales.

9.8 Considérese el esquema del Ejercicio práctico 9.2.

a. Dense instrucciones del LDD de SQL:2003 para crear la relación *EmpA*, que tiene la misma información que *Emp*, pero en la que los atributos valorados como multiconjuntos *ConjuntoNiños*, *ConjuntoConocimientos* y *ConjuntoExámenes* se sustituyen por los atributos valorados como arrays *ArrayHijos*, *ArrayConocimientos* y *ArrayExámenes*.
b. Escríbase una consulta para transformar los datos del esquema de *Emp* al de *EmpA*, con el array de los hijos ordenado por fecha de cumpleaños, el de conocimientos por el tipo de conocimientos y el de exámenes por el año de realización.
c. Escríbase una instrucción de SQL para actualizar la relación *Emp* añadiendo el hijo Jorge, con fecha de nacimiento de 5 de febrero de 2001 al empleado llamado Gabriel.
d. Escríbase una instrucción de SQL para llevar a cabo la misma actualización que antes, pero sobre la relación *EmpA*. Asegúrese de que el array de los hijos sigue ordenado por años.

9.9 Considérense los esquemas de la tabla *personas* y las tablas *estudiantes* y *profesores* que se crearon bajo *personas* en el Apartado 9.4. Dese un esquema relacional en la tercera forma normal que represente la misma información. Recuérdense las restricciones de las subtablas y dense todas las restricciones que deban imponerse en el esquema relacional para que cada ejemplar de la base de datos del esquema relacional pueda representarse también mediante un ejemplar del esquema con herencia.

9.10 Explíquese la diferencia entre el tipo x y el tipo de referencia **ref**(x). ¿En qué circunstancias se debe escoger usar el tipo de referencia?

9.11 **a.** Dese una definición de esquema en SQL:1999 del diagrama E-R de la Figura 9.7, que contiene especializaciones, usando subtipos y subtablas.
b. Dese una consulta de SQL:1999 para averiguar el nombre de todas las personas que no son secretarias.
c. Dese una consulta de SQL:1999 para imprimir el nombre de las personas que no sean empleados ni clientes.
d. ¿Se puede crear una persona que sea a la vez empleado y cliente con el esquema que se acaba de crear? Explíquese la manera de hacerlo o el motivo de que no sea posible.

9.12 Supóngase que una base de datos de JDO tiene un objeto A, que hace referencia al objeto B, que, a su vez, hace referencia al objeto C. Supóngase que todos los objetos se hallan inicialmente en el disco. Supóngase que un programa desvincula en primer lugar a A, luego a B siguiendo la referencia de A y, finalmente, desvincula a C. Muéstrense los objetos que están en representados en memoria tras cada desvinculación, junto con su estado (hueco o lleno, y los valores de los campos de referencia).

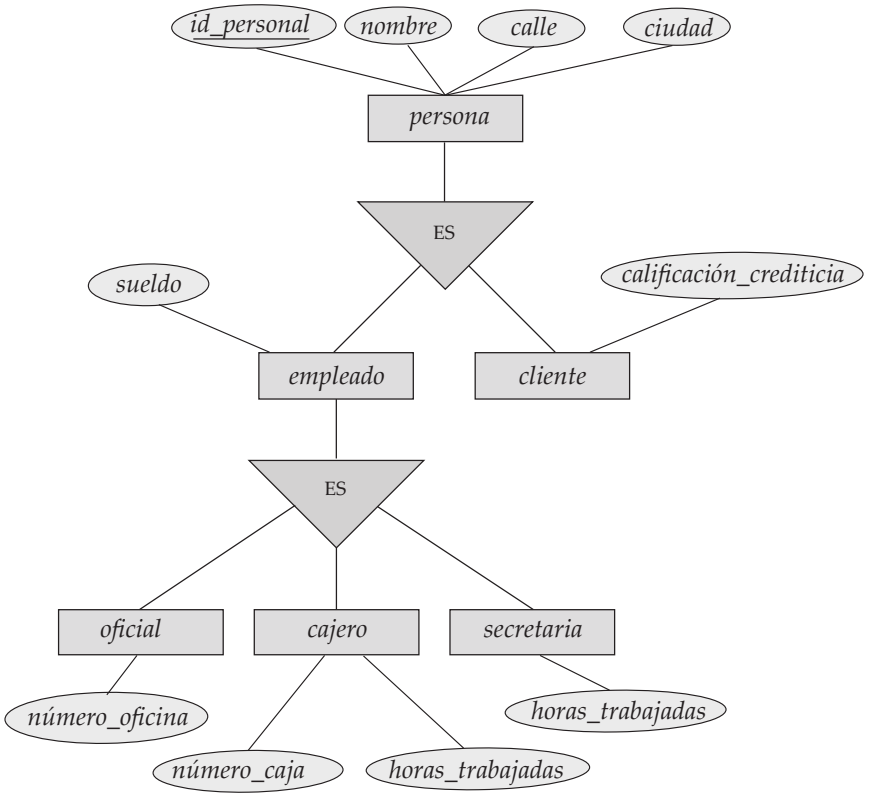

Figura 9.7 Especialización y generalización.

Notas bibliográficas

Se han propuesto varias extensiones de SQL orientadas a objetos. POSTGRES (Stonebraker y Rowe [1986] y Stonebraker [1986]) fue una de las primeras implementaciones de un sistema relacional orientado a objetos. Otros sistemas relacionales orientados a objetos de la primera época son las extensiones de SQL de O_2 (Bancilhon et al. [1989]) y UniSQL (UniSQL [1991]). SQL:1999 fue producto de un amplio (y muy tardío) esfuerzo de normalización, que se inició originalmente mediante el añadido a SQL de características orientadas a objetos y acabó añadiendo muchas más características, como las estructuras procedimentales que se han visto anteriormente. El soporte de los tipos multiconjuntos se añadió como parte de SQL:2003.

Entre los libros de texto sobre SQL:1999 están Melton y Simon [2001] y Melton [2002]; el último se centra en las características relacionales orientadas a objetos de SQL:1999. Eisenberg et al. [2004] ofrece una visión general de SQL:2003, incluido el soporte de los multiconjuntos. Se deben consultar los manuales (en línea) del sistema de bases de datos que se utilice para averiguar las características de SQL:1999 y de SQL:2003 que soporta.

A finales de los años ochenta y principios de los años noventa del siglo veinte se desarrollaron varios sistemas de bases de datos orientadas a objetos. Entre los más notables de los comerciales figuran ObjectStore (Lamb et al. [1991]), O_2 (Lecluse et al. [1988]) y Versant. La norma para bases de datos orientadas a objetos ODMG se describe con detalle en Cattell [2000]. JDO se describe en Roos [2002], Tyagi et al. [2003] y Jordan y Russell [2003].

Herramientas

Hay diferencias considerables entre los diversos productos de bases de datos en cuanto al soporte de las características relacionales orientadas a objetos. Probablemente Oracle tenga el soporte más amplio entre los principales fabricantes de bases de datos. El sistema de bases de datos de Informix ofrece so-

porte para muchas características relacionales orientadas a objetos. Tanto Oracle como Informix ofrecían características relacionales orientadas a objetos antes de la finalización de la norma SQL:1999 y presentan algunas características que no forman parte de SQL:1999.

La información sobre ObjectStore y sobre Versant, incluida la descarga de versiones de prueba, se puede obtener de sus respectivos sitios Web (objectstore.com y versant.com). El proyecto Apache DB (db.apache.org) ofrece una herramienta de asignación relacional orientada a objetos para Java que soporta tanto Java de ODMG como las APIs de JDO. Se puede obtener una implementación de referencia de JDO de sun.com; se puede usar un motor de búsqueda para averiguar el URL completo.