CHAPTER 29

Object-Based Databases

Traditional database applications consist of data-processing tasks, such as banking and payroll management, with relatively simple data types that are well suited to the relational data model. In particular, tables that are in 1NF. As database systems were applied to a wider range of applications, such as computer-aided design and geographical information systems, limitations imposed by the relational model emerged as an obstacle. The solution was the introduction of more complex data types—tables that are not in 1NF, array and multiset types, and object-based databases.

29.1 Complex Data Types

Traditional database applications have conceptually simple data types. The basic data items are records that are fairly small and whose fields are atomic—that is, they are not further structured, and first normal form holds (see Chapter 7). Further, there are only a few record types.

In recent years, demand has grown for ways to deal with more **complex data types**. Consider, for example, addresses. While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which could be of interest to queries. On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field. A better alternative is to allow **structured data types** that allow a type *address* with subparts *street_address, city, state*, and *postal_code*.

As another example, consider multivalued attributes from the E-R model. Such attributes are natural, for example, for representing phone numbers, since people may have more than one phone. The alternative of normalization by creating a new relation is expensive and artificial for this example.

With complex type systems we can represent E-R model concepts, such as composite attributes, multivalued attributes, generalization, and specialization directly, without a complex translation to the relational model.

In Chapter 7, we defined *first normal form* (1NF), which requires that all attributes have *atomic domains*. Recall that a domain is *atomic* if elements of the domain are considered to be indivisible units.

The assumption of 1NF is a natural one in the database application examples we have considered. However, not all applications are best modeled by 1NF relations. For example, rather than view a database as a set of records, users of certain applications view it as a set of objects (or entities). These objects may require several records for their representation. A simple, easy-to-use interface requires a one-to-one correspondence between the user's intuitive notion of an object and the database system's notion of a data item.

Consider, for example, a library application, and suppose we wish to store the following information for each book:

- Book title
- List of authors
- Publisher
- Set of keywords

We can see that, if we define a relation for the preceding information, several domains will be non atomic.

- **Authors**. A book may have a list of authors, which we can represent as an array. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element "authors."
- **Keywords**. If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more specified keywords. Thus, we view the domain of the set of keywords as non atomic.
- **Publisher**. Unlike *keywords* and *authors*, *publisher* does not have a set-valued domain. However, we may view *publisher* as consisting of the subfields *name* and *branch*. This view makes the domain of *publisher* non atomic.

Figure 29.1 shows an example relation, *books*.

title	*author_array*	*publisher*	*keyword_set*
		(*name, branch*)	
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web }

Figure 29.1 Non-1NF books relation, *books*.

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub_name*	*pub_branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Figure 29.2 4NF version of the relation *books*.

For simplicity, we assume that the title of a book uniquely identifies the book.[1] We can then represent the same information using the following schema, where the primary key attributes are underlined:

- *authors*(*title, author, position*)
- *keywords*(*title, keyword*)
- *books4*(*title, pub_name, pub_branch*)

The above schema satisfies 4NF. Figure 29.2 shows the normalized representation of the data from Figure 29.1.

Although our example book database can be adequately expressed without using **nested relations**, the use of nested relations leads to an easier-to-understand model. The typical user or programmer of an information-retrieval system thinks of the database in terms of books having sets of authors, as the non-1NF design models. The 4NF design requires queries to join multiple relations, whereas the non-1NF design makes many types of queries easier.

[1]This assumption does not hold in the real world. Books are usually identified by a 10-digit ISBN number that uniquely identifies each published book.

On the other hand, it may be better to use a first normal form representation in other situations. For instance, consider the *takes* relationship in our university example. The relationship is many-to-many between *student* and *section*. We could conceivably store a set of sections with each student, or a set of students with each section, or both. If we store both, we would have data redundancy (the relationship of a particular student to a particular section would be stored twice).

The ability to use complex data types such as sets and arrays can be useful in many applications but should be used with care.

29.2 SQL Extensions to Deal with Complex Data Types

Before SQL:1999, the SQL type system consisted of a fairly simple set of predefined types. SQL:1999 added an extensive type system to SQL, allowing structured types and type inheritance.

Structured types allow composite attributes of E-R designs to be represented directly. For instance, we can define the following structured type to represent a composite attribute *name* with component attribute *firstname* and *lastname*:

```
create type Name as
    (firstname varchar(20),
    lastname varchar(20))
    final;
```

Similarly, the following structured type can be used to represent a composite attribute *address*:

```
create type Address as
    (street varchar(20),
    city varchar(20),
    not final;
```

Such types are called **user-defined** types in SQL.[2] The above definition corresponds to the E-R diagram in Figure 6.7. The **final** and **not final** specifications are related to subtyping, which we describe in Section 29.3.1.[3]

We can now use these types to create composite attributes in a relation, by simply declaring an attribute to be of one of these types. For example, we could create a table *person* as follows:

[2]To illustrate our earlier note about commercial implementations defining their syntax before the standards were developed, we point out that Oracle requires the keyword **object** following **as**.

[3]The **final** specification for *Name* indicates that we cannot create subtypes for *name*, whereas the **not final** specification for *Address* indicates that we can create subtypes of *address*.

```
create table person (
        name Name,
        address Address,
        dateOfBirth date);
```

The components of a composite attribute can be accessed using a "dot" notation; for instance *name.firstname* returns the firstname component of the name attribute. An access to attribute *name* would return a value of the structured type *Name*.

We can also create a table whose rows are of a user-defined type. For example, we could define a type *PersonType* and create the table *person* as follows:[4]

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
create table person of PersonType;
```

An alternative way of defining composite attributes in SQL is to use unnamed **row types**. For instance, the relation representing person information could have been created using row types as follows:

```
create table person_r (
        name row (firstname varchar(20),
                  lastname varchar(20)),
        address row (street varchar(20),
                     city varchar(20),
        dateOfBirth date);
```

This definition is equivalent to the preceding table definition, except that the attributes *name* and *address* have unnamed types, and the rows of the table also have an unnamed type.

The following query illustrates how to access component attributes of a composite attribute. The query finds the last name and city of each person.

```
select name.lastname, address.city
from person;
```

A structured type can have **methods** defined on it. We declare methods as part of the type definition of a structured type:

[4]Most actual systems, being case insensitive, would not permit *name* to be used both as an attribute name and as a data type.

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
method ageOnDate(onDate date)
        returns interval year;
```

We create the method body separately:

```
create instance method ageOnDate (onDate date)
        returns interval year
        for PersonType
begin
        return onDate − self.dateOfBirth;
end
```

Note that the **for** clause indicates which type this method is for, while the keyword **instance** indicates that this method executes on an instance of the *Person* type. The variable **self** refers to the *Person* instance on which the method is invoked. The body of the method can contain procedural statements, which we saw in Section 5.2. Methods can update the attributes of the instance on which they are executed.

Methods can be invoked on instances of a type. If we had created a table *person* of type *PersonType*, we could invoke the method *ageOnDate*( ) as illustrated below, to find the age of each person.

```
select name.lastname, ageOnDate(current_date)
from person;
```

In SQL:1999, **constructor functions** are used to create values of structured types. A function with the same name as a structured type is a constructor function for the structured type. For instance, we could declare a constructor for the type *Name* like this:

```
create function Name (firstname varchar(20), lastname varchar(20))
returns Name
begin
        set self.firstname = firstname;
        set self.lastname = lastname;
end
```

We can then use **new** *Name*('John', 'Smith') to create a value of the type *Name*. We can construct a row value by listing its attributes within parentheses. For instance, if

we declare an attribute *name* as a row type with components *firstname* and *lastname* we can construct this value for it: ('Ted', 'Codd') without using a constructor.

By default, every structured type has a constructor with no arguments, which sets the attributes to their default values. Any other constructors have to be created explicitly. There can be more than one constructor for the same structured type; although they have the same name, they must be distinguishable by the number of arguments and types of their arguments.

The following statement illustrates how we can create a new tuple in the *Person* relation. We assume that a constructor has been defined for *Address*, just like the constructor we defined for *Name*.

```
insert into Person
    values
        (new Name('John', 'Smith'),
        new Address('20 Main St', 'New York', '11001'),
        date '1960-8-22');
```

29.3 Type and Table Inheritance

29.3.1 Type Inheritance

Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
     address varchar(20));
```

We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, we can use **inheritance** to define the student and teacher types in SQL:

```
create type Student
    under Person
    (degree varchar(20),
     department varchar(20));

create type Teacher
    under Person
    (salary integer,
     department varchar(20));
```

Both *Student* and *Teacher* inherit the attributes of *Person*—namely, *name* and *address*. *Student* and *Teacher* are said to be subtypes of *Person*, and *Person* is a supertype of *Student*, as well as of *Teacher*.

Methods of a structured type are inherited by its subtypes, just as attributes are. However, a subtype can redefine the effect of a method by declaring the method again, using **overriding method** in place of **method** in the method declaration.

The SQL standard requires an extra field at the end of the type definition, whose value is either **final** or **not final**. The keyword **final** says that subtypes may not be created from the given type, while **not final** says that subtypes may be created.

Now suppose that we want to store information about teaching assistants, who are simultaneously students and teachers, perhaps even in different departments. We can do this if the type system supports **multiple inheritance**, where a type is declared as a subtype of multiple types. Note that the SQL standard does not support multiple inheritance, although future versions of the SQL standard may support it, so we discuss the concept here.

For instance, if our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

create type *TeachingAssistant*
under *Student*, *Teacher*;

TeachingAssistant inherits all the attributes of *Student* and *Teacher*. There is a problem, however, since the attributes *name*, *address*, and *department* are present in *Student*, as well as in *Teacher*.

The attributes *name* and *address* are actually inherited from a common source, *Person*. So there is no conflict caused by inheriting them from *Student* as well as *Teacher*. However, the attribute *department* is defined separately in *Student* and *Teacher*. In fact, a teaching assistant may be a student of one department and a teacher in another department. To avoid a conflict between the two occurrences of *department*, we can rename them by using an **as** clause, as in this definition of the type *TeachingAssistant*:

create type *TeachingAssistant*
under *Student* **with** (*department* **as** *student_dept)*,
Teacher **with** (*department* **as** *teacher_dept*);

In SQL, as in most other languages, a value of a structured type must have exactly one *most-specific type.* That is, each value must be associated with one specific type, called its **most-specific type**, when it is created. By means of inheritance, it is also associated with each of the supertypes of its most-specific type. For example, suppose that an entity has the type *Person*, as well as the type *Student*. Then, the most-specific type of the entity is *Student*, since *Student* is a subtype of *Person*. However, an entity cannot have the type *Student* as well as the type *Teacher* unless it has a type, such as *Teachin-*

gAssistant, that is a subtype of *Teacher*, as well as of *Student* (which is not possible in SQL since multiple inheritance is not supported by SQL).

29.3.2 Table Inheritance

Subtables in SQL correspond to the E-R notion of specialization/generalization. For instance, suppose we define the *people* table as follows:

create table *people* **of** *Person*;

We can then define tables *students* and *teachers* as **subtables** of *people*, as follows:

create table *students* **of** *Student*
under *people*;

create table *teachers* **of** *Teacher*
under *people*;

The types of the subtables (*Student* and *Teacher* in the above example) are subtypes of the type of the parent table (*Person* in the above example). As a result, every attribute present in the table *people* is also present in the subtables *students* and *teachers*.

Further, when we declare *students* and *teachers* as subtables of *people*, every tuple present in *students* or *teachers* becomes implicitly present in *people*. Thus, if a query uses the table *people*, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely *students* and *teachers*. However, only those attributes that are present in *people* can be accessed by that query.

SQL permits us to find tuples that are in *people* but not in its subtables by using "**only** *people*" in place of *people* in a query. The **only** keyword can also be used in delete and update statements. Without the **only** keyword, a delete statement on a supertable, such as *people*, also deletes tuples that were originally inserted in subtables (such as *students*); for example, a statement:

delete from *people* **where** *P*;

would delete all tuples from the table *people*, as well as its subtables *students* and *teachers*, that satisfy *P*. If the **only** keyword is added to the above statement, tuples that were inserted in subtables are not affected, even if they satisfy the **where** clause conditions. Subsequent queries on the supertable would continue to find these tuples.

Conceptually, multiple inheritance is possible with tables, just as it is possible with types. For example, we can create a table of type *TeachingAssistant*:

create table *teaching_assistants*
of *TeachingAssistant*
under *students*, *teachers*;

As a result of the declaration, every tuple present in the *teaching_assistants* table is also implicitly present in the *teachers* and in the *students* table, and in turn in the *people* table. We note, however, that multiple inheritance of tables is not supported by SQL.

There are some consistency requirements for subtables. Before we state the constraints, we need a definition: we say that tuples in a subtable and parent table **correspond** if they have the same values for all inherited attributes. Thus, corresponding tuples represent the same entity.

The consistency requirements for subtables are:

1. Each tuple of the supertable can correspond to at most one tuple in each of its immediate subtables.
2. SQL has an additional constraint that all the tuples corresponding to each other must be derived from one tuple (inserted into one table).

For example, without the first condition, we could have two tuples in *students* (or *teachers*) that correspond to the same person.

The second condition rules out a tuple in *people* corresponding to both a tuple in *students* and a tuple in *teachers*, unless all these tuples are implicitly present because a tuple was inserted in a table *teaching_assistants*, which is a subtable of both *teachers* and *students*.

Since SQL does not support multiple inheritance, the second condition actually prevents a person from being both a teacher and a student. Even if multiple inheritance were supported, the same problem would arise if the subtable *teaching_assistants* were absent. It would be useful to model a situation where a person can be a teacher and a student, even if a common subtable *teaching_assistants* is not present. Thus, it can be useful to remove the second consistency constraint. Doing so would allow an object to have multiple types, without requiring it to have a most-specific type.

For example, suppose we again have the type *Person*, with subtypes *Student* and *Teacher*, and the corresponding table *people*, with subtables *teachers* and *students*. We can then have a tuple in *teachers* and a tuple in *students* corresponding to the same tuple in *people*. There is no need to have a type *TeachingAssistant* that is a subtype of both *Student* and *Teacher*. We need not create a type *TeachingAssistant* unless we wish to store extra attributes or redefine methods in a manner specific to people who are both students and teachers.

We note, however, that SQL unfortunately prohibits such a situation, because of consistency requirement 2. Since SQL also does not support multiple inheritance, we cannot use inheritance to model a situation where a person can be both a student and a teacher. As a result, SQL subtables cannot be used to represent overlapping specializations from the E-R model.

We can of course create separate tables to represent the overlapping specializations/generalizations without using inheritance. The process was described in Section 6.8.6.1. In the above example, we would create tables *people*, *students*, and *teachers*, with

the *students* and *teachers* tables containing the primary-key attribute of *Person* and other attributes specific to *Student* and *Teacher*, respectively. The *people* table would contain information about all persons, including students and teachers. We would then have to add appropriate referential-integrity constraints to ensure that students and teachers are also represented in the *people* table.

In other words, we can create our own improved implementation of the subtable mechanism using existing features of SQL, with some extra effort in defining the table, as well as some extra effort at query time to specify joins to access required attributes.

We note that SQL defines a privilege called **under**, which is required in order to create a subtype or subtable under another type or table. The motivation for this privilege is similar to that for the **references** privilege.

29.4 Array and Multiset Types in SQL

SQL supports two collection types: **arrays** and **multisets**; array types were added in SQL:1999, while multiset types were added in SQL:2003. Recall that a *multiset* is an unordered collection, where an element may occur multiple times. Multisets are like sets, except that a set allows each element to occur at most once.

Suppose we wish to record information about books, including a set of keywords for each book. Suppose also that we wished to store the names of authors of a book as an array; unlike elements in a multiset, the elements of an array are ordered, so we can distinguish the first author from the second author, and so on. The following example illustrates how these array and multiset-valued attributes can be defined in SQL:

```
create type Publisher as
        (name varchar(20),
         branch varchar(20));

create type Book as
        (title varchar(20),
         author_array varchar(20) array [10],
         pub_date date,
         publisher Publisher,
         keyword_set varchar(20) multiset);

create table books of Book;
```

The first statement defines a type called *Publisher* with two components: a name and a branch. The second statement defines a structured type *Book* that contains a *title*, an *author_array*, which is an array of up to 10 author names, a publication date, a publisher (of type *Publisher*), and a multiset of keywords. Finally, a table *books* containing tuples of type *Book* is created.

Note that we used an array, instead of a multiset, to store the names of authors, since the ordering of authors generally has some significance, whereas we believe that the ordering of keywords associated with a book is not significant.

In general, multivalued attributes from an E-R schema can be mapped to multiset-valued attributes in SQL; if ordering is important, SQL arrays can be used instead of multisets.

29.4.1 Creating and Accessing Collection Values

An array of values can be created in SQL:1999 in this way:

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

Similarly, a multiset of keywords can be constructed as follows:

multiset['computer', 'database', 'SQL']

Thus, we can create a tuple of the type defined by the *books* relation as:

('Compilers', **array**['Smith', 'Jones'], **new** *Publisher*('McGraw-Hill', 'New York'), **multiset**['parsing', 'analysis'])

Here we have created a value for the attribute *Publisher* by invoking a *constructor* function for *Publisher* with appropriate arguments. Note that this constructor for *Publisher* must be created explicitly and is not present by default; it can be declared just like the constructor for *Name*, which we saw in Section 29.2.

If we want to insert the preceding tuple into the relation *books*, we can execute the statement:

```
insert into books
values ('Compilers', array['Smith', 'Jones'],
                     new Publisher('McGraw-Hill', 'New York'),
                     multiset['parsing', 'analysis']);
```

We can access or update elements of an array by specifying the array index, for example *author_array*[1].

29.4.2 Querying Collection-Valued Attributes

We now consider how to handle collection-valued attributes in queries. An expression evaluating to a collection can appear anywhere that a relation name may appear, such as in a **from** clause, as the following paragraphs illustrate. We use the table *books* that we defined earlier.

If we want to find all books that have the word "database" as one of their keywords, we can use this query:

```
select title
from books
where 'database' in (unnest(keyword_set));
```

Note that we have used **unnest**(*keyword_set*) in a position where SQL without nested relations would have required a **select-from-where** subexpression.

If we know that a particular book has three authors, we could write:

```
select author_array[1], author_array[2], author_array[3]
from books
where title = 'Database System Concepts';
```

Now, suppose that we want a relation containing pairs of the form "title, author_name" for each book and each author of the book. We can use this query:

```
select B.title, A.author
from books as B, unnest(B.author_array) as A(author);
```

Since the *author_array* attribute of *books* is a collection-valued field, **unnest**(*B.author_array*) can be used in a **from** clause, where a relation is expected. Note that the tuple variable *B* is visible to this expression since it is defined *earlier* in the **from** clause.

When unnesting an array, the previous query loses information about the ordering of elements in the array. The **unnest with ordinality** clause can be used to get this information, as illustrated by the following query. This query can be used to generate the *authors* relation, which we saw earlier, from the *books* relation.

```
select title, A.author, A.position
from books as B,
        unnest(B.author_array) with ordinality as A(author, position);
```

The **with ordinality** clause generates an extra attribute which records the position of the element in the array. A similar query, but without the **with ordinality** clause, can be used to generate the *keyword* relation.

29.4.3 Nesting and Unnesting

The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**. The *books* relation has two attributes, *author_array* and *keyword_set*, that are collections, and two attributes, *title* and *publisher*, that are not. Suppose that we want to convert the relation into a single flat relation, with no nested relations or structured types as attributes. We can use the following query to carry out the task:

title	*author*	*pub_name*	*pub_branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Figure 29.3 *flat_books*: result of unnesting attributes *author_array* and *keyword_set* of relation *books*.

```
select title, A.author, publisher.name as pub_name, publisher.branch
        as pub_branch, K.keyword
from books as B, unnest(B.author_array) as A(author),
        unnest (B.keyword_set) as K(keyword);
```

The variable *B* in the **from** clause is declared to range over *books*. The variable *A* is declared to range over the authors in *author_array* for the book *B*, and *K* is declared to range over the keywords in the *keyword_set* of the book *B*. Figure 29.1 shows an instance of the *books* relation, and Figure 29.3 shows the relation, which we call *flat_books*, that is the result of the preceding query. Note that the relation *flat_books* is in 1NF, since all its attributes are atomic valued.

The reverse process of transforming a 1NF relation into a nested relation is called **nesting**. Nesting can be carried out by an extension of grouping in SQL. In the normal use of grouping in SQL, a temporary multiset relation is (logically) created for each group, and an aggregate function is applied on the temporary relation to get a single (atomic) value. The **collect** function returns the multiset of values, so instead of creating a single value, we can create a nested relation. Suppose that we are given the 1NF relation *flat_books*, as in Figure 29.3. The following query nests the relation on the attribute *keyword*:

```
select title, author, Publisher(pub_name, pub_branch) as publisher,
        collect(keyword) as keyword_set
from flat_books
group by title, author, publisher;
```

The result of the query on the *flat_books* relation from Figure 29.3 appears in Figure 29.4.

If we want to nest the author attribute also into a multiset, we can use the query:

title	*author*	*publisher* (*pub_name*,*pub_branch*)	*keyword_set*
Compilers	Smith	(McGraw-Hill, NewYork)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

Figure 29.4 A partially nested version of the *flat_books* relation.

```
select title, collect(author) as author_set,
       Publisher(pub_name, pub_branch) as publisher,
            collect(keyword) as keyword_set
from flat_books
group by title, publisher;
```

Another approach to creating nested relations is to use subqueries in the **select** clause. An advantage of the subquery approach is that an **order by** clause can be used in the subquery to generate results in the order desired for the creation of an array. The following query illustrates this approach; the keywords **array** and **multiset** specify that an array and multiset (respectively) are to be created from the results of the subqueries.

```
select title,
       array( select author
         from authors as A
         where A.title = B.title
         order by A.position) as author_array,
       Publisher(pub_name, pub_branch) as publisher,
       multiset( select keyword
         from keywords as K
         where K.title = B.title) as keyword_set,
from books4 as B;
```

The system executes the nested subqueries in the **select** clause for each tuple generated by the **from** and **where** clauses of the outer query. Observe that the attribute *B.title* from the outer query is used in the nested queries, to ensure that only the correct sets of authors and keywords are generated for each title.

SQL:2003 provides a variety of operators on multisets, including a function **set**(*M*) that returns a duplicate-free version of a multiset *M*, an **intersection** aggregate operation, which returns the intersection of all the multisets in a group, a **fusion** aggregate operation, which returns the union of all multisets in a group, and a **submultiset** predicate, which checks if a multiset is contained in another multiset.

The SQL standard does not provide any way to update multiset attributes except by assigning a new value. For example, to delete a value v from a multiset attribute A, we would have to set it to (A **except all multiset**[v]).

29.5 Summary

- Collection types include nested relations, sets, multisets, and arrays, and the object-relational model permits attributes of a table to be collections.
- The SQL standard includes extensions of the SQL data-definition and query language to deal with new data types and with object orientation. These include support for collection-valued attributes, inheritance, and tuple references. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power.

Review Terms

- Nested relations
- Nested relational model
- Complex types
- Collection types
- Sets
- Arrays
- Multisets
- Structured types
- Row types
- Constructors
- Inheritance
  - Single inheritance
  - Multiple inheritance
- Type inheritance
- Most-specific type
- Table inheritance
- Subtable
- Overlapping subtables
- Reference types
- Scope of a reference
- Self-referential attribute
- Path expressions
- Nesting and unnesting
- SQL functions and procedures
- Object-relational mapping

Practice Exercises

29.1 A car-rental company maintains a database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles:

- Trucks: cargo capacity
- Sports cars: horsepower, renter age requirement
- Vans: number of passengers
- Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive)

Construct an SQL schema definition for this database. Use inheritance where appropriate.

29.2 Consider a database schema with a relation *Emp* whose attributes are as shown below, with types specified for multivalued attributes.

Emp = (ename, ChildrenSet **multiset***(Children), SkillSet* **multiset***(Skills))*
Children = (name, birthday)
Skills = (type, ExamSet **setof***(Exams))*
Exams = (year, city)

Answer the following:

a. Define the above schema in SQL, with appropriate types for each attribute.

b. Using the above schema, write the following queries in SQL.

i. Find the names of all employees who have a child born on or after January 1, 2000.

ii. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

iii. List all skill types in the relation *Emp*.

29.3 Consider the E-R diagram in Figure 29.5, which contains composite, multivalued, and derived attributes.

a. Give an SQL schema definition corresponding to the E-R diagram.

b. Give constructors for each of the structured types defined above.

29.4 Consider the relational schema shown in Figure 29.6.

a. Give a schema definition in SQL corresponding to the relational schema, but using references to express foreign-key relationships.

b. Write each of the queries below on the schema in Figure 29.6, using SQL.

i. Find the company with the most employees.

ii. Find the company with the smallest payroll.

iii. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

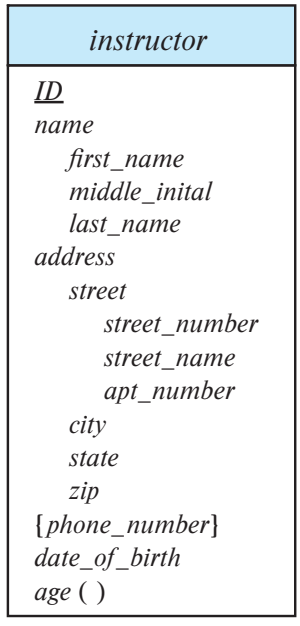

Figure 29.5 E-R diagram with composite, multivalued, and derived attributes.

29.5 Suppose that you have been hired as a consultant to choose a database system for your client's application. For each of the following applications, state what type of database system (relational, persistent programming language-based OODB, object-relational; do not specify a commercial product) you would recommend. Justify your recommendation.

a. A computer-aided design system for a manufacturer of airplanes.

b. A system to track contributions made to candidates for public office.

c. An information system to support the making of movies.

29.6 How does the concept of an object in the object-oriented model differ from the concept of an entity in the entity-relationship model?

---

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*person_name*, *manager_name*)

Figure 29.6 Relational database for Exercise 29.4.

Exercises

29.7 Redesign the database of Exercise 29.2 into first normal form and fourth normal form. List any functional or multivalued dependencies that you assume. Also list all referential-integrity constraints that should be present in the first and fourth normal form schemas.

29.8 Consider the schema from Exercise 29.2.

a. Give SQL DDL statements to create a relation *EmpA* which has the same information as *Emp*, but where multiset-valued attributes *ChildrenSet*, *SkillsSet* and *ExamsSet* are replaced by array-valued attributes *ChildrenArray*, *SkillsArray* and *ExamsArray*.

b. Write a query to convert data from the schema of *Emp* to that of *EmpA*, with the array of children sorted by birthday, the array of skills by the skill type, and the array of exams by the year.

c. Write an SQL statement to update the *Emp* relation by adding a child Jeb, with a birthdate of February 5, 2001, to the employee named George.

d. Write an SQL statement to perform the same update as above but on the *EmpA* relation. Make sure that the array of children remains sorted by year.

29.9 Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 29.3.2. Give a relational schema in third normal form that represents the same information. Recall the constraints on subtables, and give all constraints that must be imposed on the relational schema so that every database instance of the relational schema can also be represented by an instance of the schema with inheritance.

29.10 Explain the distinction between a type x and a reference type **ref**(x). Under what circumstances would you choose to use a reference type?

29.11 Consider the E-R diagram in Figure 29.7, which contains specializations, using subtypes and subtables.

a. Give an SQL schema definition of the E-R diagram.

b. Give an SQL query to find the names of all people who are not secretaries.

c. Give an SQL query to print the names of people who are neither employees nor students.

d. Can you create a person who is an employee and a student with the schema you created? Explain how, or explain why it is not possible.

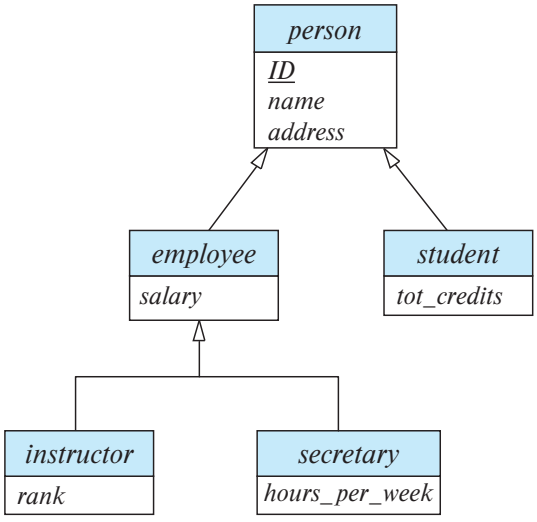

Figure 29.7 Specialization and generalization.

29.12 Suppose a JDO database had an object A, which references object B, which in turn references object C. Assume all objects are on disk initially. Suppose a program first dereferences A, then dereferences B by following the reference from A, and then finally dereferences C. Show the objects that are represented in memory after each dereference, along with their state (hollow or filled, and values in their reference fields).

Tools

There are considerable differences between database products in their support for object-relational features. Oracle probably has the most extensive support among the major database vendors. The Informix database system provides support for many object-relational features. Both Oracle and Informix provided object-relational features before the SQL:1999 standard was finalized, and they have some features that are not part of SQL:1999.

Further Reading

Several object-oriented extensions to SQL have been proposed. POSTGRES ([Stonebraker and Rowe (1986)] and [Stonebraker (1986)]) was an early implementation of an object-relational system.

Bibliography

[Stonebraker (1986)] M. Stonebraker, "Inclusion of New Types in Relational Database Systems", In *Proc. of the International Conf. on Data Engineering* (1986), pages 262–269.

[Stonebraker and Rowe (1986)] M. Stonebraker and L. Rowe, "The Design of POSTGRES", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986), pages 340–355.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.