CHAPTER 22

# Object-Based Databases

This chapter describes extensions to relational database systems to provide complex data types and object-oriented features. Such extended systems are called object-relational systems. Since the chapter was introduced in the $3^{th}$ edition most commercial database systems have added some support for object-relational features, and these features have been standardized as part of SQL:1999.

It would be instructive to assign students exercises aimed at finding applications where the object-relational model, in particular complex objects, would be better suited than the traditional relational model.

## Exercises

**22.7** Redesign the database of Practice Exercise 22.2 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.

**Answer:** To put the schema into first normal form, we flatten all the attributes into a single relation schema.

$$\mathit{Employee\text{-}details} = (\mathit{ename}, \mathit{cname}, \mathit{bday}, \mathit{bmonth}, \mathit{byear}, \mathit{stype}, \mathit{xyear}, \mathit{xcity})$$

We rename the attributes for the sake of clarity. *cname* is *Children.name*, and *bday, bmonth, byear* are the *Birthday* attributes. *stype* is *Skills.type*, and *xyear* and *xcity* are the *Exams* attributes. The FDs and multivalued dependencies we assume are:-

$$\begin{aligned} ename, cname &\rightarrow bday, bmonth, byear \\ ename &\rightarrow\!\!\!\rightarrow cname, bday, bmonth, byear \\ ename, stype &\rightarrow\!\!\!\rightarrow xyear, xcity \end{aligned}$$

The FD captures the fact that a child has a unique birthday, under the assumption that one employee cannot have two children of the same

name. The MVDs capture the fact there is no relationship between the children of an employee and his or her skills-information.
The redesigned schema in fourth normal form is:-

*Employee* = (*ename*)
*Child* = (*ename, cname, bday, bmonth, byear*)
*Skill* = (*ename, stype, xyear, xcity*)

*ename* will be the primary key of *Employee*, and (*ename, cname*) will be the primary key of *Child*. The *ename* attribute is a foreign key in *Child* and in *Skill*, referring to the *Employee* relation.

**22.8** Consider the schema from Practice Exercise 22.2.

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.

**Answer:**

a.

```
create type Exams (year int, city varchar(30))
create type SkillsA (type varchar(30),
    ExamsArray Exams array [20])
create type Children (name varchar(30), birthday date)
create table EmpA (ename varchar(30),
    ChildrenArray Children array [10],
    SkillArray SkillsA array [25])
```

b.

```
select ename,
    array(select name, birthday
         from unnest(E.ChildrenSet) as CS
         order by CS.birthday) as ChildrenArray,
    array(select type,
        array(select year, city
                     from unnest(SS.ExamSet)
                     order by SS.year) as ExamsArray
                 from unnest(E.SkillSet) as SS)
                as SkillsArray
    from Emp as E
```

c.

```
    update Emp
    set ChildrenSet = ChildrenSet union
        multiset[(''Jeb'', ''2/5/2001'')]
    where ename = ''George''
```

d. We make use of the infix array concatenation operator, "||", which takes two arrays as arguments and returns the concatenation of the two arrays.

```
    update EmpA
    set ChildrenArray = array(
        select name, birthday
        from unnest(ChildrenArray ||
            array[(''Jeb'', ''2/5/2001'')])
        order by birthday)
    where ename = ''George''
```

**22.9** Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 22.4. 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.

**Answer:** A corresponding relational schema in third normal form is given below:-

*People* = (*name*, *address*)
*Students* = (*name*, *degree*, *student-department*)
*Teachers* = (*name*, *salary*, *teacher-department*)

*name* is the primary key for all the three relations, and it is also a foreign key referring to *People*, for both *Students* and *Teachers*.
Instead of placing only the *name* attribute of *People* in *Students* and *Teachers*, both its attributes can be included. In that case, there will be a slight change, namely – (*name*, *address*) will become the foreign key in *Students* and *Teachers*. The primary keys will remain the same in all tables.

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

**Answer:** If the type of an attribute is $x$, then in each tuple of the table, corresponding to that attribute, there is an actual object of type $x$ . If its type is **ref**($x$), then in each tuple, corresponding to that attribute, there is a *reference* to some object of type $x$. We choose a reference type for an attribute, if that attribute's intended purpose is to refer to an independent object.

**22.11** Consider the E-R diagram in Figure 22.7, 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.

**Answer:**

a.
```
create type Person
    ID varchar(10),
    name varchar(30),
    address varchar(40))
create type Employee
    under Person
    (salary integer)
create type Student
    under Person
    (tot_credits integer)
create type Instructor
    under Employee
    (rank varchar(10))
create type Secretary
    under Employee
    (hours_per_week integer)

create table person of Person
create table employee of Employee
    under person
create table student of Student
    under person
create table instructor of Instructor
    under employee
```

**create table** *secretary* **of** *Secretary*
  **under** *employee*

b.

**select** *
**from** *person*
**where** *ID* **not in**
  (**select** *ID* **from** *secretary*)

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

**select** *
**from only** *person*

d. It is not possible to create a person who is an employee and a student, since there is no most-specific type in our schema corresponding to such a person. To do so, we would have to create a corresponding type (such as TeachingAssistant) using multiple inheritance, and a corresponding table that is under employee and student. However, SQL does not permit multiple inheritance of tables, so this is not possible in SQL.

**22.12** 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).

**Answer:** See figures 22.1 through 22.3. Gray boxes indicate persistent objects and white boxes indicate hollow objects.

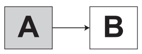

**Figure 22.1** State of the program after A is referenced.

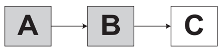

**Figure 22.2** State of the program after B is referenced.

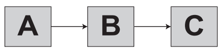

**Figure 22.3** State of the program after C is referenced.