CHAPTER 22

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. 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 object-based databases, which allow one to deal with complex data types.

22.1 Overview

The first obstacle faced by programmers using the relational data model was the limited type system supported by the relational model. Complex application domains require correspondingly complex data types, such as nested record structures, multivalued attributes, and inheritance, which are supported by traditional programming languages. Such features are in fact supported in the E-R and extended E-R notations, but had to be translated to simpler SQL data types. The **object-relational data model** extends the relational data model by providing a richer type system including complex data types and object orientation. Relational query languages, in particular SQL, need to be correspondingly extended to deal with the richer type system. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power. **Object-relational database systems**, that is, database systems based on the object-relation model, provide a convenient migration path for users of relational databases who wish to use object-oriented features.

The second obstacle was the difficulty in accessing database data from programs written in programming languages such as C++ or Java. Merely extending the type system supported by the database was not enough to solve this problem completely. Differences between the type system of the database and the type system of the programming language make data storage and retrieval more complicated, and need to be minimized. Having to express database access using a language (SQL) that is different from the programming language again makes the job of the programmer harder. It is desirable, for many applications, to have

programming language constructs or extensions that permit direct access to data in the database, without having to go through an intermediate language such as SQL.

In this chapter, we first explain the motivation for the development of complex data types. We then study object-relational database systems, specifically using features that were introduced in SQL:1999 and SQL:2003. Note that most database products support only a subset of the SQL features described here and for supported features, the syntax often differs slightly from the standard. This is the result of commercial systems introducing object-relational features to the market before the standards were finalized. Refer to the user manual of the database system you use to find out what features it supports.

We then address the issue of supporting persistence for data that is in the native type system of an object-oriented programming language. Two approaches are used in practice:

1. Build an **object-oriented database system**, that is, a database system that natively supports an object-oriented type system, and allows direct access to data from an object-oriented programming language using the native type system of the language.
2. Automatically convert data from the native type system of the programming language to a relational representation, and vice versa. Data conversion is specified using an **object-relational mapping**.

We provide a brief introduction to both these approaches.

Finally, we outline situations in which the object-relational approach is better than the object-oriented approach, and vice versa, and mention criteria for choosing between them.

22.2 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 8). 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

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

Figure 22.1 Non-1NF books relation, *books*.

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 8, 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 nonatomic.

- **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 nonatomic.
- **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* nonatomic.

Figure 22.1 shows an example 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 22.2 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 22.2 shows the normalized representation of the data from Figure 22.1.

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.

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

[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.

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.

22.3 Structured Types and Inheritance in SQL

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.

22.3.1 Structured Types

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),
    zipcode varchar(9))
    not final;
```

Such types are called **user-defined** types in SQL[2]. The above definition corresponds to the E-R diagram in Figure 7.4. The **final** and **not final** specifications are related to subtyping, which we describe later, in Section 22.3.2.[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),
                     zipcode varchar(9)),
        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 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 earlier 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');
```

22.3.2 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 *TeachingAssistant*, 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).

22.4 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. Obviously 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 earlier, in Section 7.8.6.1. 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.

22.5 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.

22.5.1 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 earlier in Section 22.3.

If we want to insert the preceding tuple into the relation *books*, we could 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].

22.5.2 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.

22.5.3 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 22.3 *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 22.1 shows an instance *books* relation, and Figure 22.3 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 22.3. 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 22.3 appears in Figure 22.4.

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, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

Figure 22.4 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]).

22.6 Object-Identity and Reference Types in SQL

Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL we can define a type *Department* with a field *name* and a field *head* that is a reference to the type *Person*, and a table *departments* of type *Department*, as follows:

```
create type Department (
        name varchar(20),
        head ref(Person) scope people);

create table departments of Department;
```

Here, the reference is restricted to tuples of the table *people*. The restriction of the **scope** of a reference to tuples of a table is mandatory in SQL, and it makes references behave like foreign keys.

We can omit the declaration **scope** *people* from the type declaration and instead make an addition to the **create table** statement:

```
create table departments of Department
        (head with options scope people);
```

The referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called the **self-referential attribute**, by adding a **ref is** clause to the **create table** statement:

```
create table people of Person
ref is person_id system generated;
```

Here, *person_id* is an attribute name, not a keyword, and the **create table** statement specifies that the identifier is generated automatically by the database.

In order to initialize a reference attribute, we need to get the identifier of the tuple that is to be referenced. We can get the identifier value of a tuple by means of a query. Thus, to create a tuple with the reference value, we may first create the tuple with a null reference and then set the reference separately:

```
insert into departments
    values ('CS', null);

update departments
    set head = (select p.person_id
                from people as p
                where name = 'John')
where name = 'CS';
```

An alternative to system-generated identifiers is to allow users to generate identifiers. The type of the self-referential attribute must be specified as part of the type definition of the referenced table, and the table definition must specify that the reference is **user generated**:

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

create table people of Person
    ref is person_id user generated;
```

When inserting a tuple in *people*, we must then provide a value for the identifier:

```
insert into people (person_id, name, address) values
    ('01284567', 'John', '23 Coyote Run');
```

No other tuple for *people* or its supertables or subtables can have the same identifier. We can then use the identifier value when inserting a tuple into *departments*, without the need for a separate query to retrieve the identifier:

```
insert into departments
    values ('CS', '01284567');
```

It is even possible to use an existing primary-key value as the identifier, by including the **ref from** clause in the type definition:

```
create type Person
    (name varchar(20) primary key,
     address varchar(20))
    ref from(name);

create table people of Person
    ref is person_id derived;
```

Note that the table definition must specify that the reference is derived, and must still specify a self-referential attribute name. When inserting a tuple for *departments*, we can then use:

```
insert into departments
    values ('CS', 'John');
```

References are dereferenced in SQL:1999 by the −> symbol. Consider the *departments* table defined earlier. We can use this query to find the names and addresses of the heads of all departments:

```
select head−>name, head−>address
from departments;
```

An expression such as "*head*−>*name*" is called a **path expression**.

Since *head* is a reference to a tuple in the *people* table, the attribute *name* in the preceding query is the *name* attribute of the tuple from the *people* table. References can be used to hide join operations; in the preceding example, without the references, the *head* field of *department* would be declared a foreign key of the table *people*. To find the name and address of the head of a department, we would require an explicit join of the relations *departments* and *people*. The use of references simplifies the query considerably.

We can use the operation **deref** to return the tuple pointed to by a reference, and then access its attributes, as shown below:

```
select deref(head).name
from departments;
```

22.7 Implementing O-R Features

Object-relational database systems are basically extensions of existing relational database systems. Changes are clearly required at many levels of the database system. However, to minimize changes to the storage-system code (relation storage, indices, etc.), the complex data types supported by object-relational systems can be translated to the simpler type system of relational databases.

To understand how to do this translation, we need look only at how some features of the E-R model are translated into relations. For instance, multivalued attributes in the E-R model correspond to multiset-valued attributes in the object-relational model. Composite attributes roughly correspond to structured types. ISA hierarchies in the E-R model correspond to table inheritance in the object-relational model.

The techniques for converting E-R model features to tables, which we saw in Section 7.6, can be used, with some extensions, to translate object-relational data to relational data at the storage level.

Subtables can be stored in an efficient manner, without replication of all inherited fields, in one of two ways:

- Each table stores the primary key (which may be inherited from a parent table) and the attributes that are defined locally. Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the supertable, based on the primary key.
- Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the supertables. Access to all attributes of a tuple is faster, since a join is not required.

  However, in case the type system allows an entity to be represented in two subtables without being present in a common subtable of both, this representation can result in replication of information. Further, it is hard to translate foreign keys referring to a supertable into constraints on the subtables; to implement such foreign keys efficiently, the supertable has to be defined as a view, and the database system would have to support foreign keys on views.

Implementations may choose to represent array and multiset types directly, or may choose to use a normalized representation internally. Normalized representations tend to take up more space and require an extra join/grouping cost to collect data in an array or multiset. However, normalized representations may be easier to implement.

The ODBC and JDBC application program interfaces have been extended to retrieve and store structured types. JDBC provides a method `getObject()` that is similar to `getString()` but returns a Java `Struct` object, from which the components of the structured type can be extracted. It is also possible to associate a Java class with an SQL structured type, and JDBC will then convert between the types. See the ODBC or JDBC reference manuals for details.

22.8 Persistent Programming Languages

Database languages differ from traditional programming languages in that they directly manipulate data that are persistent—that is, data that continue to exist even after the program that created it has terminated. A relation in a database and tuples in a relation are examples of persistent data. In contrast, the only persistent data that traditional programming languages directly manipulate are files.

Access to a database is only one component of any real-world application. While a data-manipulation language like SQL is quite effective for accessing data, a programming language is required for implementing other components of the application such as user interfaces or communication with other computers. The traditional way of interfacing database languages to programming languages is by embedding SQL within the programming language.

A **persistent programming language** is a programming language extended with constructs to handle persistent data. Persistent programming languages can be distinguished from languages with embedded SQL in at least two ways:

1. With an embedded language, the type system of the host language usually differs from the type system of the data-manipulation language. The programmer is responsible for any type conversions between the host language and SQL. Having the programmer carry out this task has several drawbacks:
   - The code to convert between objects and tuples operates outside the object-oriented type system, and hence has a higher chance of having undetected errors.
   - Conversion between the object-oriented format and the relational format of tuples in the database takes a substantial amount of code. The format translation code, along with the code for loading and unloading data from a database, can form a significant percentage of the total code required for an application.

   In contrast, in a persistent programming language, the query language is fully integrated with the host language, and both share the same type system. Objects can be created and stored in the database without any explicit type or format changes; any format changes required are carried out transparently.
2. The programmer using an embedded query language is responsible for writing explicit code to fetch data from the database into memory. If any updates are performed, the programmer must write code explicitly to store the updated data back in the database.

   In contrast, in a persistent programming language, the programmer can manipulate persistent data without writing code explicitly to fetch it into memory or store it back to disk.

In this section, we describe how object-oriented programming languages, such as C++ and Java, can be extended to make them persistent programming languages. These language features allow programmers to manipulate data directly from the programming language, without having to go through a data-manipulation language such as SQL. They thereby provide tighter integration of the programming languages with the database than, for example, embedded SQL.

There are certain drawbacks to persistent programming languages, however, that we must keep in mind when deciding whether to use them. Since the programming language is usually a powerful one, it is relatively easy to make programming errors that damage the database. The complexity of the language makes automatic high-level optimization, such as to reduce disk I/O, harder. Support for declarative querying is important for many applications, but persistent programming languages currently do not support declarative querying well.

In this section, we describe a number of conceptual issues that must be addressed when adding persistence to an existing programming language. We first

address language-independent issues, and in subsequent sections we discuss issues that are specific to the C++ language and to the Java language. However, we do not cover details of language extensions; although several standards have been proposed, none has met universal acceptance. See the references in the bibliographical notes to learn more about specific language extensions and further details of implementations.

22.8.1 Persistence of Objects

Object-oriented programming languages already have a concept of objects, a type system to define object types, and constructs to create objects. However, these objects are *transient*—they vanish when the program terminates, just as variables in a Java or C program vanish when the program terminates. If we wish to turn such a language into a database programming language, the first step is to provide a way to make objects persistent. Several approaches have been proposed.

- **Persistence by class**. The simplest, but least convenient, way is to declare that a class is persistent. All objects of the class are then persistent objects by default. Objects of nonpersistent classes are all transient.

  This approach is not flexible, since it is often useful to have both transient and persistent objects in a single class. Many object-oriented database systems interpret declaring a class to be persistent as saying that objects in the class potentially can be made persistent, rather than that all objects in the class are persistent. Such classes might more appropriately be called "persistable" classes.

- **Persistence by creation**. In this approach, new syntax is introduced to create persistent objects, by extending the syntax for creating transient objects. Thus, an object is either persistent or transient, depending on how it was created. Several object-oriented database systems follow this approach.

- **Persistence by marking**. A variant of the preceding approach is to mark objects as persistent after they are created. All objects are created as transient objects, but, if an object is to persist beyond the execution of the program, it must be marked explicitly as persistent before the program terminates. This approach, unlike the previous one, postpones the decision on persistence or transience until after the object is created.

- **Persistence by reachability**. One or more objects are explicitly declared as (root) persistent objects. All other objects are persistent if (and only if) they are reachable from the root object through a sequence of one or more references.

  Thus, all objects referenced by (that is, whose object identifiers are stored in) the root persistent objects are persistent. But also, all objects referenced from these objects are persistent, and objects to which they refer are in turn persistent, and so on.

  A benefit of this scheme is that it is easy to make entire data structures persistent by merely declaring the root of such structures as persistent. How-

ever, the database system has the burden of following chains of references to detect which objects are persistent, and that can be expensive.

22.8.2 Object Identity and Pointers

In an object-oriented programming language that has not been extended to handle persistence, when an object is created, the system returns a transient object identifier. Transient object identifiers are valid only when the program that created them is executing; after that program terminates, the objects are deleted, and the identifier is meaningless. When a persistent object is created, it is assigned a persistent object identifier.

The notion of object identity has an interesting relationship to pointers in programming languages. A simple way to achieve built-in identity is through pointers to physical locations in storage. In particular, in many object-oriented languages such as C++, a transient object identifier is actually an in-memory pointer.

However, the association of an object with a physical location in storage may change over time. There are several degrees of permanence of identity:

- **Intraprocedure**. Identity persists only during the execution of a single procedure. Examples of intraprogram identity are local variables within procedures.
- **Intraprogram**. Identity persists only during the execution of a single program or query. Examples of intraprogram identity are global variables in programming languages. Main-memory or virtual-memory pointers offer only intraprogram identity.
- **Interprogram**. Identity persists from one program execution to another. Pointers to file-system data on disk offer interprogram identity, but they may change if the way data is stored in the file system is changed.
- **Persistent**. Identity persists not only among program executions, but also among structural reorganizations of the data. It is the persistent form of identity that is required for object-oriented systems.

In persistent extensions of languages such as C++, object identifiers for persistent objects are implemented as "persistent pointers." A *persistent pointer* is a type of pointer that, unlike in-memory pointers, remains valid even after the end of a program execution, and across some forms of data reorganization. A programmer may use a persistent pointer in the same ways that she may use an in-memory pointer in a programming language. Conceptually, we may think of a persistent pointer as a pointer to an object in the database.

22.8.3 Storage and Access of Persistent Objects

What does it mean to store an object in a database? Clearly, the data part of an object has to be stored individually for each object. Logically, the code that

implements methods of a class should be stored in the database as part of the database schema, along with the type definitions of the classes. However, many implementations simply store the code in files outside the database, to avoid having to integrate system software such as compilers with the database system.

There are several ways to find objects in the database. One way is to give names to objects, just as we give names to files. This approach works for a relatively small number of objects, but does not scale to millions of objects. A second way is to expose object identifiers or persistent pointers to the objects, which can be stored externally. Unlike names, these pointers do not have to be mnemonic, and they can even be physical pointers into a database.

A third way is to store collections of objects, and to allow programs to iterate over the collections to find required objects. Collections of objects can themselves be modeled as objects of a *collection type*. Collection types include sets, multisets (that is, sets with possibly many occurrences of a value), lists, and so on. A special case of a collection is a **class extent**, which is the collection of all objects belonging to the class. If a class extent is present for a class, then, whenever an object of the class is created, that object is inserted in the class extent automatically, and, whenever an object is deleted, that object is removed from the class extent. Class extents allow classes to be treated like relations in that we can examine all objects in the class, just as we can examine all tuples in a relation.

Most object-oriented database systems support all three ways of accessing persistent objects. They give identifiers to all objects. They usually give names only to class extents and other collection objects, and perhaps to other selected objects, but not to most objects. They usually maintain class extents for all classes that can have persistent objects, but, in many of the implementations, the class extents contain only persistent objects of the class.

22.8.4 Persistent C++ Systems

There are several object-oriented databases based on persistent extensions to C++ (see the bibliographical notes). There are differences among them in terms of the system architecture, yet they have many common features in terms of the programming language.

Several of the object-oriented features of the C++ language provide support for persistence without changing the language itself. For example, we can declare a class called `Persistent_Object` with attributes and methods to support persistence; any other class that should be persistent can be made a subclass of this class, and thereby inherit the support for persistence. The C++ language (like some other modern programming languages) also lets us redefine standard function names and operators—such as +, −, the pointer dereference operator −>, and so on—according to the types of the operands on which they are applied. This ability is called *overloading*; it is used to redefine operators to behave in the required manner when they are operating on persistent objects.

Providing persistence support via class libraries has the benefit of making only minimal changes to C++ necessary; moreover, it is relatively easy to implement. However, it has the drawback that the programmer has to spend much more

time to write a program that handles persistent objects, and it is not easy for the programmer to specify integrity constraints on the schema or to provide support for declarative querying. Some persistent C++ implementations support extensions to the C++ syntax to make these tasks easier.

The following aspects need to be addressed when adding persistence support to C++ (and other languages):

- **Persistent pointers**: A new data type has to be defined to represent persistent pointers. For example, the ODMG C++ standard defines a template class `d_Ref`$< T >$ to represent persistent pointers to a class T. The dereference operator on this class is redefined to fetch the object from disk (if not already present in memory), and it returns an in-memory pointer to the buffer where the object has been fetched. Thus if p is a persistent pointer to a class T, one can use standard syntax such as `p->A` or `p->f(v)` to access attribute A of class T or invoke method f of class T.

  The ObjectStore database system uses a different approach to persistent pointers. It uses normal pointer types to store persistent pointers. This poses two problems: (1) in-memory pointer sizes may be only 4 bytes, which is too small to use with databases larger than 4 gigabytes, and (2) when an object is moved on disk, in-memory pointers to its old physical location are meaningless. ObjectStore uses a technique called "hardware swizzling" to address both problems; it prefetches objects from the database into memory, and replaces persistent pointers with in-memory pointers, and when data are stored back on disk, in-memory pointers are replaced by persistent pointers. When on disk, the value stored in the in-memory pointer field is not the actual persistent pointer; instead, the value is looked up in a table that contains the full persistent pointer value.

- **Creation of persistent objects**: The C++ `new` operator is used to create persistent objects by defining an "overloaded" version of the operator that takes extra arguments specifying that it should be created in the database. Thus instead of `new T()`, one would call `new (db) T()` to create a persistent object, where `db` identifies the database.

- **Class extents**: Class extents are created and maintained automatically for each class. The ODMG C++ standard requires the name of the class to be passed as an additional parameter to the `new` operation. This also allows multiple extents to be maintained for a class, by passing different names.

- **Relationships**: Relationships between classes are often represented by storing pointers from each object to the objects to which it is related. Objects related to multiple objects of a given class store a set of pointers. Thus if a pair of objects is in a relationship, each should store a pointer to the other. Persistent C++ systems provide a way to specify such integrity constraints and to enforce them by automatically creating and deleting pointers: For example, if a pointer is created from an object a to an object b, a pointer to a is added automatically to object b.

- **Iterator interface**: Since programs need to iterate over class members, an interface is required to iterate over members of a class extent. The iterator interface also allows selections to be specified, so that only objects satisfying the selection predicate need to be fetched.
- **Transactions**: Persistent C++ systems provide support for starting a transaction, and for committing it or rolling it back.
- **Updates**: One of the goals of providing persistence support in a programming language is to allow transparent persistence. That is, a function that operates on an object should not need to know that the object is persistent; the same functions can thus be used on objects regardless of whether they are persistent or not.

  However, one resultant problem is that it is difficult to detect when an object has been updated. Some persistent extensions to C++ require the programmer to specify explicitly that an object has been modified by calling a function `mark_modified()`. In addition to increasing programmer effort, this approach increases the chance that programming errors can result in a corrupt database. If a programmer omits a call to `mark_modified()`, it is possible that one update made by a transaction may never be propagated to the database, while another update made by the same transaction is propagated, violating atomicity of transactions.

  Other systems, such as ObjectStore, use memory-protection support provided by the operating system/hardware to detect writes to a block of memory and mark the block as a dirty block that should be written later to disk.
- **Query language**: Iterators provide support for simple selection queries. To support more complex queries, persistent C++ systems define a query language.

A large number of object-oriented database systems based on C++ were developed in the late 1980s and early 1990s. However, the market for such databases turned out to be much smaller than anticipated, since most application requirements are more than met by using SQL through interfaces such as ODBC or JDBC. As a result, most of the object-oriented database systems developed in that period do not exist any longer. In the 1990s, the Object Data Management Group (ODMG) defined standards for adding persistence to C++ and Java. However, the group wound up its activities around 2002. ObjectStore and Versant are among the original object-oriented database systems that are still in existence.

Although object-oriented database systems did not find the commercial success that they had hoped for, the motivation for adding persistence to programming language remains. There are several applications with high performance requirements that run on object-oriented database systems; using SQL would impose too high a performance overhead for many such systems. With object-relational database systems now providing support for complex data types, including references, it is easier to store programming language objects in an SQL

database. A new generation of object-oriented database systems using object-relational databases as a backend may yet emerge.

22.8.5 Persistent Java Systems

The Java language has seen an enormous growth in usage in recent years. Demand for support for persistence of data in Java programs has grown correspondingly. Initial attempts at creating a standard for persistence in Java were led by the ODMG consortium; the consortium wound up its efforts later, but transferred its design to the **Java Database Objects** (**JDO**) effort, which is coordinated by Sun Microsystems.

The JDO model for object persistence in Java programs differs from the model for persistence support in C++ programs. Among its features are:

- **Persistence by reachability**: Objects are not explicitly created in a database. Explicitly registering an object as persistent (using the `makePersistent()` method of the `PersistenceManager` class) makes the object persistent. In addition, any object reachable from a persistent object becomes persistent.
- **Byte code enhancement**: Instead of declaring a class to be persistent in the Java code, classes whose objects may be made persistent are specified in a configuration file (with suffix `.jdo`). An implementation-specific *enhancer* program is executed that reads the configuration file and carries out two tasks. First, it may create structures in a database to store objects of the class. Second, it modifies the byte code (generated by compiling the Java program) to handle tasks related to persistence. Below are some examples of such modifications:
  - Any code that accesses an object could be changed to check first if the object is in memory, and if not, take steps to bring it into memory.
  - Any code that modifies an object is modified to record additionally that the object has been modified, and perhaps to save a pre-updated value used in case the update needs to be undone (that is, if the transaction is rolled back).

  Other modifications to the byte code may also be carried out. Such byte code modification is possible since the byte code is standard across all platforms, and includes much more information than compiled object code.
- **Database mapping**: JDO does not define how data are stored in the back-end database. For example, a common scenario is to store objects in a relational database. The enhancer program may create an appropriate schema in the database to store class objects. How exactly it does this is implementation dependent and not defined by JDO. Some attributes could be mapped to relational attributes, while others may be stored in a serialized form, treated as a binary object by the database. JDO implementations may allow existing relational data to be viewed as objects by defining an appropriate mapping.

- **Class extents**: Class extents are created and maintained automatically for each class declared to be persistent. All objects made persistent are added automatically to the class extent corresponding to their class. JDO programs may access a class extent, and iterate over selected members. The `Iterator` interface provided by Java can be used to create iterators on class extents, and to step through the members of the class extent. JDO also allows selections to be specified when an iterator is created on a class extent, and only objects satisfying the selection are fetched.

- **Single reference type**: There is no difference in type between a reference to a transient object and a reference to a persistent object.

  One approach to achieving such a unification of pointer types would be to load the entire database into memory, replacing all persistent pointers with in-memory pointers. After updates were done, the process would be reversed, storing updated objects back on disk. Such an approach would be very inefficient for large databases.

  We now describe an alternative approach that allows persistent objects to be fetched automatically into memory when required, while allowing all references contained in in-memory objects to be in-memory references. When an object A is fetched, a **hollow object** is created for each object B_i that it references, and the in-memory copy of A has references to the corresponding hollow object for each B_i. Of course the system has to ensure that if an object B_i was fetched already, the reference points to the already fetched object instead of creating a new hollow object. Similarly, if an object B_i has not been fetched, but is referenced by another object fetched earlier, it would already have a hollow object created for it; the reference to the existing hollow object is reused, instead of creating a new hollow object.

  Thus, for every object O_i that has been fetched, every reference from O_i is either to an already fetched object or to a hollow object. The hollow objects form a *fringe* surrounding fetched objects.

  Whenever the program actually accesses a hollow object O, the enhanced byte code detects this and fetches the object from the database. When this object is fetched, the same process of creating hollow objects is carried out for all objects referenced by O. After this the access to the object is allowed to proceed.[5]

  An in-memory index structure mapping persistent pointers to in-memory references is required to implement this scheme. In writing objects back to disk, this index would be used to replace in-memory references with persistent pointers in the copy written to disk.

[5]The technique using hollow objects described above is closely related to the hardware swizzling technique (mentioned earlier in Section 22.8.4). Hardware swizzling is used by some persistent C++ implementations to provide a single pointer type for persistent and in-memory pointers. Hardware swizzling uses virtual-memory protection techniques provided by the operating system to detect accesses to pages, and fetches the pages from the database when required. In contrast, the Java version modifies byte code to check for hollow objects, instead of using memory protection, and fetches objects when required, instead of fetching whole pages from the database.

22.9 Object-Relational Mapping

So far we have seen two approaches to integrating object-oriented data models and programming languages with database systems. **Object-relational mapping** systems provide a third approach to integration of object-oriented programming languages and databases.

Object-relational mapping systems are built on top of a traditional relational database, and allow a programmer to define a mapping between tuples in database relations and objects in the programming language. Unlike in persistent programming languages, objects are transient, and there is no permanent object identity.

An object, or a set of objects, can be retrieved based on a selection condition on its attributes; relevant data are retrieved from the underlying database based on the selection conditions, and one or more objects are created from the retrieved data, based on the prespecified mapping between objects and relations. The program can optionally update such objects, create new objects, or specify that an object is to be deleted, and then issue a save command; the mapping from objects to relations is then used to correspondingly update, insert or delete tuples in the database.

Object-relational mapping systems in general, and in particular the widely used Hibernate system which provides an object-relational mapping to Java, are described in more detail in Section 9.4.2.

The primary goal of object-relational mapping systems is to ease the job of programmers who build applications, by providing them an object-model, while retaining the benefits of using a robust relational database underneath. As an added benefit, when operating on objects cached in memory, object-relational systems can provide significant performance gains over direct access to the underlying database.

Object-relational mapping systems also provide query languages that allow programmers to write queries directly on the object model; such queries are translated into SQL queries on the underlying relational database, and result objects created from the SQL query results.

On the negative side, object-relational mapping systems can suffer from significant overheads for bulk database updates, and may provide only limited querying capabilities. However, it is possible to directly update the database, bypassing the object-relational mapping system, and to write complex queries directly in SQL. The benefits or object-relational models exceed the drawbacks for many applications, and object-relational mapping systems have seen widespread adoption in recent years.

22.10 Object-Oriented versus Object-Relational

We have now studied object-relational databases, which are object-oriented databases built on top of the relation model, as well as object-oriented databases, which are built around persistent programming languages, and object-relational

mapping systems, which build an object layer on top of a traditional relational database.

Each of these approaches targets a different market. The declarative nature and limited power (compared to a programming language) of the SQL language provides good protection of data from programming errors, and makes high-level optimizations, such as reducing I/O, relatively easy. (We covered optimization of relational expressions in Chapter 13.) Object-relational systems aim at making data modeling and querying easier by using complex data types. Typical applications include storage and querying of complex data, including multimedia data.

A declarative language such as SQL, however, imposes a significant performance penalty for certain kinds of applications that run primarily in main memory, and that perform a large number of accesses to the database. Persistent programming languages target such applications that have high performance requirements. They provide low-overhead access to persistent data and eliminate the need for data translation if the data are to be manipulated by a programming language. However, they are more susceptible to data corruption by programming errors, and they usually do not have a powerful querying capability. Typical applications include CAD databases.

Object-relational mapping systems allow programmers to build applications using an object model, while using a traditional database system to store the data. Thus, they combine the robustness of widely used relational database systems, with the power of object models for writing applications. However, they suffer from overheads of data conversion between the object model and the relational model used to store data.

We can summarize the strengths of the various kinds of database systems in this way:

- **Relational systems**: Simple data types, powerful query languages, high protection.
- **Persistent programming language–based OODBs**: Complex data types, integration with programming language, high performance.
- **Object-relational systems**: Complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**: Complex data types integrated with programming languages, designed as a layer on top of a relational database system.

These descriptions hold in general, but keep in mind that some database systems blur the boundaries. For example, object-oriented database systems built around a persistent programming language can be implemented on top of a relational or object-relational database system. Such systems may provide lower performance than object-oriented database systems built directly on a storage system, but provide some of the stronger protection guarantees of relational systems.

22.11 Summary

- The object-relational data model extends the relational data model by providing a richer type system including collection types and object orientation.
- Collection types include nested relations, sets, multisets, and arrays, and the object-relational model permits attributes of a table to be collections.
- Object orientation provides inheritance with subtypes and subtables, as well as object (tuple) references.
- 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.
- Object-relational database systems (that is, database systems based on the object-relation model) provide a convenient migration path for users of relational databases who wish to use object-oriented features.
- Persistent extensions to C++ and Java integrate persistence seamlessly and orthogonally with existing programming language constructs and so are easy to use.
- The ODMG standard defines classes and other constructs for creating and accessing persistent objects from C++, while the JDO standard provides equivalent functionality for Java.
- Object-relational mapping systems provide an object view of data that is stored in a relational database. Objects are transient, and there is no notion of persistent object identity. Objects are created on-demand from relational data, and updates to objects are implemented by updating the relational data. Object-relational mapping systems have been widely adopted, unlike the more limited adoption of persistent programming languages.
- We discussed differences between persistent programming languages and object-relational systems, and we mention criteria for choosing between them.

Review Terms

- Nested relations
- Nested relational model
- Complex types
- Collection types
- Large object types
- Sets
- Arrays
- Multisets
- Structured types
- Methods