5. Codd, E. F., "Further normalization of the data base relational model," in *Database Systems* (R. Rustin, ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972.

6. Delobel, C., "Normalization and hierarchical dependencies in the relational data model," *ACM Transactions on Database Systems* **3**:3, pp. 201–222, 1978.

7. Fagin, R., "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems* **2**:3, pp. 262–278, 1977.

8. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

9. Zaniolo, C. and M. A. Melkanoff, "On the design of relational database schemata," *ACM Transactions on Database Systems* **6**:1, pp. 1–47, 1981.

Chapter 4

Other Data Models

The entity-relationship and relational models are just two of the models that have importance in database systems today. In this chapter we shall introduce you to several other models of rising importance.

We begin with a discussion of object-oriented data models. One approach to object-orientation for a database system is to extend the concepts of object-oriented programming languages such as C++ or Java to include persistence. That is, the presumption in ordinary programming is that objects go away after the program finishes, while an essential requirement of a DBMS is that the objects are preserved indefinitely, unless changed by the user, as in a file system. We shall study a "pure" object-oriented data model, called ODL (object definition language), which has been standardized by the ODMG (object data management group).

Next, we consider a model called object-relational. This model, part of the most recent SQL standard, called SQL-99 (or SQL:1999, or SQL3), is an attempt to extend the relational model, as introduced in Chapter 3, to include many of the common object-oriented concepts. This standard forms the basis for object-relational DBMS's that are now available from essentially all the major vendors, although these vendors differ considerably in the details of how the concepts are implemented and made available to users. Chapter 9 includes a discussion of the object-relational model of SQL-99.

Then, we take up the "semistructured" data model. This recent innovation is an attempt to deal with a number of database problems, including the need to combine databases and other data sources, such as Web pages, that have different schemas. While an essential of object-oriented or object-relational systems is their insistence on a fixed schema for every class or every relation, semistructured data is allowed much more flexibility in what components are present. For instance, we could think of movie objects, some of which have a director listed, some of which might have several different lengths for several different versions, some of which may include textual reviews, and so on.

The most prominent implementation of semistructured data is XML (exten-

sible markup language). Essentially, XML is a specification for "documents," which are really collections of nested data elements, each with a role indicated by a tag. We believe that XML data will serve as an essential component in systems that mediate among data sources or that transmit data among sources. XML may even become an important approach to flexible storage of data in databases.

4.1 Review of Object-Oriented Concepts

Before introducing object-oriented database models, let us review the major object-oriented concepts themselves. Object-oriented programming has been widely regarded as a tool for better program organization and, ultimately, more reliable software implementation. First popularized in the language Smalltalk, object-oriented programming received a big boost with the development of C++ and the migration to C++ of much software development that was formerly done in C. More recently, the language Java, suitable for sharing programs across the World Wide Web, has also focused attention on object-oriented programming.

The database world has likewise been attracted to the object-oriented paradigm, particularly for database design and for extending relational DBMS's with new features. In this section we shall review the ideas behind object orientation:

1. A powerful type system.
2. *Classes*, which are types associated with an *extent*, or set of *objects* belonging to the class. An essential feature of classes, as opposed to conventional data types is that classes may include *methods*, which are procedures that are applicable to objects belonging to the class.
3. *Object Identity*, the idea that each object has a unique identity, independent of its value.
4. *Inheritance*, which is the organization of classes into hierarchies, where each class inherits the properties of the classes above it.

4.1.1 The Type System

An object-oriented programming language offers the user a rich collection of types. Starting with *atomic types*, such as integers, real numbers, booleans, and character strings, one may build new types by using *type constructors*. Typically, the type constructors let us build:

1. *Record structures.* Given a list of types $T_1, T_2, \ldots, T_n$ and a corresponding list of *field names* (called *instance variables* in Smalltalk) $f_1, f_2, \ldots, f_n$, one can construct a record type consisting of n components. The ith component has type T_i and is referred to by its field name f_i. Record structures are exactly what C or C++ calls "structs," and we shall frequently use that term in what follows.

2. *Collection types.* Given a type T, one can construct new types by applying a *collection operator* to type T. Different languages use different collection operators, but there are several common ones, including arrays, lists, and sets. Thus, if T were the atomic type integer, we might build the collection types "array of integers," "list of integers," or "set of integers."
3. *Reference types.* A reference to a type T is a type whose values are suitable for locating a value of the type T. In C or C++, a reference is a "pointer" to a value, that is, the virtual-memory address of the value pointed to.

Of course, record-structure and collection operators can be applied repeatedly to build ever more complex types. For instance, a bank might define a type that is a record structure with a first component named `customer` of type string and whose second component is of type set-of-integers and is named `accounts`. Such a type is suitable for associating bank customers with the set of their accounts.

4.1.2 Classes and Objects

A *class* consists of a type and possibly one or more functions or procedures (called *methods*; see below) that can be executed on objects of that class. The objects of a class are either values of that type (called *immutable objects*) or variables whose value is of that type (called *mutable objects*). For example, if we define a class C whose type is "set of integers," then $\{2, 5, 7\}$ is an immutable object of class C, while variable s could be declared to be a mutable object of class C and assigned a value such as $\{2, 5, 7\}$.

4.1.3 Object Identity

Objects are assumed to have an *object identity* (OID). No two objects can have the same OID, and no object has two different OID's. Object identity has some interesting effects on how we model data. For instance, it is essential that an entity set have a key formed from values of attributes possessed by it or a related entity set (in the case of weak entity sets). However, within a class, we assume we can distinguish two objects whose attributes all have identical values, because the OID's of the two objects are guaranteed to be different.

4.1.4 Methods

Associated with a class there are usually certain functions, often called *methods*. A method for a class C has at least one argument that is an object of class C; it may have other arguments of any class, including C. For example, associated

with a class whose type is "set of integers," we might have methods to sum the elements of a given set, to take the union of two sets, or to return a boolean indicating whether or not the set is empty.

In some situations, classes are referred to as "abstract data types," meaning that they *encapsulate*, or restrict access to objects of the class so that only the methods defined for the class can modify objects of the class directly. This restriction assures that the objects of the class cannot be changed in ways that were not anticipated by the designer of the class. Encapsulation is regarded as one of the key tools for reliable software development.

4.1.5 Class Hierarchies

It is possible to declare one class C to be a *subclass* of another class D. If so, then class C *inherits* all the properties of class D, including the type of D and any functions defined for class D. However, C may also have additional properties. For example, new methods may be defined for objects of class C, and these methods may be either in addition to or in place of methods of D. It may even be possible to extend the type of D in certain ways. In particular, if the type of D is a record-structure type, then we can add new fields to this type that are present only in objects of type C.

Example 4.1: Consider a class of bank account objects. We might describe the type for this class informally as:

```
CLASS Account = {accountNo: integer;
                 balance: real;
                 owner: REF Customer;
                }
```

That is, the type for the `Account` class is a record structure with three fields: an integer account number, a real-number balance, and an owner that is a reference to an object of class `Customer` (another class that we'd need for a banking database, but whose type we have not introduced here).

We could also define some methods for the class. For example, we might have a method

```
deposit(a: Account, m: real)
```

that increases the `balance` for `Account` object a by amount m.

Finally, we might wish to have several subclasses of the `Account` subclass. For instance, a time-deposit account could have an additional field `dueDate`, the date at which the account balance may be withdrawn by the owner. There might also be an additional method for the subclass `TimeDeposit`

```
penalty(a: TimeDeposit)
```


that takes an account a belonging to the subclass `TimeDeposit` and calculates the penalty for early withdrawal, as a function of the `dueDate` field in object a and the current date; the latter would be obtainable from the system on which the method is run. □

4.2 Introduction to ODL

ODL (Object Definition Language) is a standardized language for specifying the structure of databases in object-oriented terms. It is an extension of IDL (Interface Description Language), a component of CORBA (Common Object Request Broker Architecture). The latter is a standard for distributed, object-oriented computing.

4.2.1 Object-Oriented Design

In an object-oriented design, the world to be modeled is thought of as composed of *objects*, which are observable entities of some sort. For example, people may be thought of as objects; so may bank accounts, airline flights, courses at a college, buildings, and so on. Objects are assumed to have a unique *object identity* (OID) that distinguishes them from any other object, as we discussed in Section 4.1.3.

To organize information, we usually want to group objects into *classes* of objects with similar properties. However, when speaking of ODL object-oriented designs, we should think of "similar properties" of the objects in a class in two different ways:

- The real-world concepts represented by the objects of a class should be similar. For instance, it makes sense to group all customers of a bank into one class and all accounts at the bank into another class. It would not make sense to group customers and accounts together in one class, because they have little or nothing in common and play essentially different roles in the world of banking.

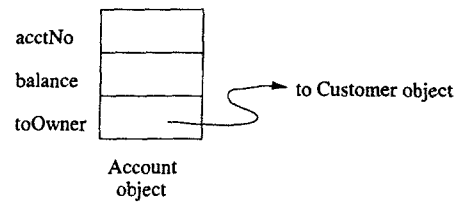

Figure 4.1: An object representing an account

- The properties of objects in a class must be the same. When programming in an object-oriented language, we often think of objects as records, like

that suggested by Fig. 4.1. Objects have fields or slots in which values are placed. These values may be of common types such as integers, strings, or arrays, or they may be references to other objects.

When specifying the design of ODL classes, we describe properties of three kinds:

1. *Attributes*, which are values associated with the object. We discuss the legal types of ODL attributes in Section 4.2.8.
2. *Relationships*, which are connections between the object at hand and another object or objects.
3. *Methods*, which are functions that may be applied to objects of the class.

Attributes, relationships, and methods are collectively referred to as *properties*.

4.2.2 Class Declarations

A declaration of a class in ODL, in its simplest form, consists of:

1. The keyword `class`,
2. The name of the class, and
3. A bracketed list of properties of the class. These properties can be attributes, relationships, or methods, mixed in any order.

That is, the simple form of a class declaration is

```
class <name> {
    <list of properties>
}
```

4.2.3 Attributes in ODL

The simplest kind of property is the *attribute*. These properties describe some aspect of an object by associating a value of a fixed type with that object. For example, person objects might each have an attribute `name` whose type is string and whose value is the name of that person. Person objects might also have an attribute `birthdate` that is a triple of integers (i.e., a record structure) representing the year, month, and day of their birth.

In ODL, unlike the E/R model, attributes need not be of simple types, such as integers and strings. We just mentioned `birthdate` as an example of an attribute with a structured type. For another example, an attribute such as `phones` might have a set of strings as its type, and even more complex types are possible. We summarize the type system of ODL in Section 4.2.8.

Example 4.2: In Fig. 4.2 is an ODL declaration of the class of movies. It is not a complete declaration; we shall add more to it later. Line (1) declares `Movie` to be a class. Following line (1) are the declarations of four attributes that all `Movie` objects will have.

```
1)  class Movie {
2)      attribute string title;
3)      attribute integer year;
4)      attribute integer length;
5)      attribute enum Film {color,blackAndWhite} filmType;
    };
```

Figure 4.2: An ODL declaration of the class `Movie`

The first attribute, on line (2), is named `title`. Its type is `string`—a character string of unknown length. We expect the value of the `title` attribute in any `Movie` object to be the name of the movie. The next two attributes, `year` and `length` declared on lines (3) and (4), have integer type and represent the year in which the movie was made and its length in minutes, respectively. On line (5) is another attribute `filmType`, which tells whether the movie was filmed in color or black-and-white. Its type is an *enumeration*, and the name of the enumeration is `Film`. Values of enumeration attributes are chosen from a list of *literals*, `color` and `blackAndWhite` in this example.

An object in the class `Movie` as we have defined it so far can be thought of as a record or tuple with four components, one for each of the four attributes. For example,

```
("Gone With the Wind", 1939, 231, color)
```

is a `Movie` object. □

Example 4.3: In Example 4.2, all the attributes have atomic types. Here is an example with a nonatomic type. We can define the class `Star` by

```
1)  class Star {
2)      attribute string name;
3)      attribute Struct Addr
            {string street, string city} address;
    };
```

Line (2) specifies an attribute `name` (of the star) that is a string. Line (3) specifies another attribute `address`. This attribute has a type that is a *record structure*. The name of this structure is `Addr`, and the type consists of two fields: `street` and `city`. Both fields are strings. In general, one can define record structure types in ODL by the keyword `Struct` and curly braces around

Why Name Enumerations and Structures?

The name `Film` for the enumeration on line 5 of Fig. 4.2 doesn't seem to be necessary. However, by giving it a name, we can refer to it outside the scope of the declaration for class `Movie`. We do so by referring to it by the *scoped name* `Movie::Film`. For instance, in a declaration of a class of cameras, we could have a line:

```
attribute Movie::Film uses;
```

This line declares attribute `uses` to be of the same enumerated type with the values `color` and `blackAndWhite`.

Another reason for giving names to enumerated types (and structures as well, which are declared in a manner similar to enumerations) is that we can declare them in a "module" outside the declaration of any particular class, and have that type available to all the classes in the module.

the list of field names and their types. Like enumerations, structure types must have a name, which can be used elsewhere to refer to the same structure type. □

4.2.4 Relationships in ODL

While we can learn much about an object by examining its attributes, sometimes a critical fact about an object is the way it connects to other objects in the same or another class.

Example 4.4: Now, suppose we want to add to the declaration of the `Movie` class from Example 4.2 a property that is a set of stars. More precisely, we want each `Movie` object to connect the set of `Star` objects that are its stars. The best way to represent this connection between the `Movie` and `Star` classes is with a *relationship*. We may represent this relationship in `Movie` by a line:

```
relationship Set<Star> stars;
```

in the declaration of class `Movie`. This line may appear in Fig. 4.2 after any of the lines numbered (1) through (5). It says that in each object of class `Movie` there is a set of references to `Star` objects. The set of references is called `stars`. The keyword `relationship` specifies that `stars` contains references to other objects, while the keyword `Set` preceding `<Star>` indicates that `stars` references a set of `Star` objects, rather than a single object. In general, a type that is a set of elements of some other type T is defined in ODL by the keyword `Set` and angle brackets around the type T. □

4.2.5 Inverse Relationships

Just as we might like to access the stars of a given movie, we might like to know the movies in which a given star acted. To get this information into `Star` objects, we can add the line

```
relationship Set<Movie> starredIn;
```

to the declaration of class `Star` in Example 4.3. However, this line and a similar declaration for `Movie` omits a very important aspect of the relationship between movies and stars. We expect that if a star S is in the `stars` set for movie M, then movie M is in the `starredIn` set for star S. We indicate this connection between the relationships `stars` and `starredIn` by placing in each of their declarations the keyword `inverse` and the name of the other relationship. If the other relationship is in some other class, as it usually is, then we refer to that relationship by the name of its class, followed by a double colon (`::`) and the name of the relationship.

Example 4.5: To define the relationship `starredIn` of class `Star` to be the inverse of the relationship `stars` in class `Movie`, we revise the declarations of these classes, as shown in Fig. 4.3 (which also contains a definition of class `Studio` to be discussed later). Line (6) shows the declaration of relationship `stars` of movies, and says that its inverse is `Star::starredIn`. Since relationship `starredIn` is defined in another class, the relationship name is preceded by the name of that class (`Star`) and a double colon. Recall the double colon is used whenever we refer to something defined in another class, such as a property or type name.

Similarly, relationship `starredIn` is declared in line (11). Its inverse is declared by that line to be `stars` of class `Movie`, as it must be, because inverses always are linked in pairs. □

As a general rule, if a relationship R for class C associates with object x of class C with objects $y_1, y_2, \ldots, y_n$ of class D, then the inverse relationship of R associates with each of the y_i's the object x (perhaps along with other objects). Sometimes, it helps to visualize a relationship R from class C to class D as a list of pairs, or tuples, of a relation. The idea is the same as the "relationship set" we used to describe E/R relationships in Section 2.1.5. Each pair consists of an object x from class C and an associated object y of class D, as:

C	D
x_1	y_1
x_2	y_2
...	...

Then the inverse relationship for R is the set of pairs with the components reversed, as:

```
1)  class Movie {
2)      attribute string title;
3)      attribute integer year;
4)      attribute integer length;
5)      attribute enum Film {color,blackAndWhite} filmType;
6)      relationship Set<Star> stars
                    inverse Star::starredIn;
7)      relationship Studio ownedBy
                    inverse Studio::owns;
    };

8)  class Star {
9)      attribute string name;
10)     attribute Struct Addr
            {string street, string city} address;
11)     relationship Set<Movie> starredIn
                    inverse Movie::stars;
    };

12) class Studio {
13)     attribute string name;
14)     attribute string address;
15)     relationship Set<Movie> owns
                    inverse Movie::ownedBy;
    };
```

Figure 4.3: Some ODL classes and their relationships

D	C
y_1	x_1
y_2	x_2
...	...

Notice that this rule works even if C and D are the same class. There are some relationships that logically run from a class to itself, such as "child of" from the class "Persons" to itself.

4.2.6 Multiplicity of Relationships

Like the binary relationships of the E/R model, a pair of inverse relationships in ODL can be classified as either many-many, many-one in either direction, or one-one. The type declarations for the pair of relationships tells us which.

1. If we have a many-many relationship between classes C and D, then in class C the type of the relationship is `Set<`D`>`, and in class D the type is `Set<`C`>`.[1]
2. If the relationship is many-one from C to D, then the type of the relationship in C is just D, while the type of the relationship in D is `Set<`C`>`.
3. If the relationship is many-one from D to C, then the roles of C and D are reversed in (2) above.
4. If the relationship is one-one, then the type of the relationship in C is just D, and in D it is just C.

Note, that as in the E/R model, we allow a many-one or one-one relationship to include the case where for some objects the "one" is actually "none." For instance, a many-one relationship from C to D might have a missing or "null" value of the relationship in some of the C objects. Of course, since a D object could be associated with any set of C objects, it is also permissible for that set to be empty for some D objects.

Example 4.6: In Fig. 4.3 we have the declaration of three classes, `Movie`, `Star`, and `Studio`. The first two of these have already been introduced in Examples 4.2 and 4.3. We also discussed the relationship pair `stars` and `starredIn`. Since each of their types uses `Set`, we see that this pair represents a many-many relationship between `Star` and `Movie`.

Studio objects have attributes `name` and `address`; these appear in lines (13) and (14). Notice that the type of addresses here is a string, rather than a structure as was used for the `address` attribute of class `Star` on line (10). There is nothing wrong with using attributes of the same name but different types in different classes.

In line (7) we see a relationship `ownedBy` from movies to studios. Since the type of the relationship is `Studio`, and not `Set<Studio>`, we are declaring that for each movie there is one studio that owns it. The inverse of this relationship is found on line (15). There we see the relationship `owns` from studios to movies. The type of this relationship is `Set<Movie>`, indicating that each studio owns a set of movies—perhaps 0, perhaps 1, or perhaps a large number of movies. □

4.2.7 Methods in ODL

The third kind of property of ODL classes is the method. As in other object-oriented languages, a method is a piece of executable code that may be applied to the objects of the class.

In ODL, we can declare the names of the methods associated with a class and the input/output types of those methods. These declarations, called *signatures*,

[1] Actually, the `Set` could be replaced by another "collection type," such as list or bag, as discussed in Section 4.2.8. We shall assume all collections are sets in our exposition of relationships, however.

Why Signatures?

The value of providing signatures is that when we implement the schema in a real programming language, we can check automatically that the implementation matches the design as was expressed in the schema. We cannot check that the implementation correctly implements the "meaning" of the operations, but we can at least check that the input and output parameters are of the correct number and of the correct type.

are like function declarations in C or C++ (as opposed to function *definitions*, which are the code to implement the function). The code for a method would be written in the host language; this code is not part of ODL.

Declarations of methods appear along with the attributes and relationships in a class declaration. As is normal for object-oriented languages, each method is associated with a class, and methods are invoked on an object of that class. Thus, the object is a "hidden" argument of the method. This style allows the same method name to be used for several different classes, because the object upon which the operation is performed determines the particular method meant. Such a method name is said to be *overloaded.*

The syntax of method declarations is similar to that of function declarations in C, with two important additions:

1. Method parameters are specified to be `in`, `out`, or `inout`, meaning that they are used as input parameters, output parameters, or both, respectively. The last two types of parameters can be modified by the method; `in` parameters cannot be modified. In effect, `out` and `inout` parameters are passed by reference, while `in` parameters may be passed by value. Note that a method may also have a return value, which is a way that a result can be produced by a method other than by assigning a value to an `out` or `inout` parameter.

2. Methods may raise *exceptions*, which are special responses that are outside the normal parameter-passing and return-value mechanisms by which methods communicate. An exception usually indicates an abnormal or unexpected condition that will be "handled" by some method that called it (perhaps indirectly through a sequence of calls). Division by zero is an example of a condition that might be treated as an exception. In ODL, a method declaration can be followed by the keyword `raises`, followed by a parenthesized list of one or more exceptions that the method can raise.

Example 4.7: In Fig. 4.4 we see an evolution of the definition for class `Movie`, last seen in Fig. 4.3. The methods included with the class declaration are as follows.

Line (8) declares a method `lengthInHours`. We might imagine that it produces as a return value the length of the movie object to which it is applied, but converted from minutes (as in the attribute `length`) to a floating-point number that is the equivalent in hours. Note that this method takes no parameters. The `Movie` object to which the method is applied is the "hidden" argument, and it is from this object that a possible implementation of `lengthInHours` would obtain the length of the movie in minutes.[2]

Method `lengthInHours` may raise an exception called `noLengthFound`. Presumably this exception would be raised if the `length` attribute of the object to which the method `lengthInHours` was applied had an undefined value or a value that could not represent a valid length (e.g., a negative number).

```
1)  class Movie {
2)      attribute string title;
3)      attribute integer year;
4)      attribute integer length;
5)      attribute enumeration(color,blackAndWhite) filmType;
6)      relationship Set<Star> stars
                    inverse Star::starredIn;
7)      relationship Studio ownedBy
                    inverse Studio::owns;
8)      float lengthInHours() raises(noLengthFound);
9)      void starNames(out Set<String>);
10)     void otherMovies(in Star, out Set<Movie>)
                    raises(noSuchStar);
    };
```

Figure 4.4: Adding method signatures to the `Movie` class

In line (9) we see another method signature, for a method called `starNames`. This method has no return value but has an output parameter whose type is a set of strings. We presume that the value of the output parameter is computed by `starNames` to be the set of strings that are the values of the attribute `name` for the stars of the movie to which the method is applied. However, as always there is no guarantee that the method definition behaves in this particular way.

Finally, at line (10) is a third method, `otherMovies`. This method has an input parameter of type `Star`. A possible implementation of this method is as follows. We may suppose that `otherMovies` expects this star to be one of the stars of the movie; if it is not, then the exception `noSuchStar` is raised. If it is one of the stars of the movie to which the method is applied, then the output parameter, whose type is a set of movies, is given as its value the set of all the

[2] In the actual definition of the method `lengthInHours` a special term such as `self` would be used to refer to the object to which the method is applied. This matter is of no concern as far as declarations of method signatures is concerned.

other movies of this star. □

4.2.8 Types in ODL

ODL offers the database designer a type system similar to that found in C or other conventional programming languages. A type system is built from a basis of types that are defined by themselves and certain recursive rules whereby complex types are built from simpler types. In ODL, the basis consists of:

1. *Atomic types*: integer, float, character, character string, boolean, and *enumerations*. The latter are lists of names declared to be abstract values. We saw an example of an enumeration in line (5) of Fig. 4.3, where the names are `color` and `blackAndWhite`.

2. *Class names*, such as `Movie`, or `Star`, which represent types that are actually structures, with components for each of the attributes and relationships of that class.

These basic types are combined into structured types using the following *type constructors*:

1. *Set.* If T is any type, then `Set<T>` denotes the type whose values are finite sets of elements of type T. Examples using the set type-constructor occur in lines (6), (11), and (15) of Fig. 4.3.

2. *Bag.* If T is any type, then `Bag<T>` denotes the type whose values are finite bags or *multisets* of elements of type T. A bag allows an element to appear more than once. For example, $\{1,2,1\}$ is a bag but not a set, because 1 appears more than once.

3. *List.* If T is any type, then `List<T>` denotes the type whose values are finite lists of zero or more elements of type T. As a special case, the type `string` is a shorthand for the type `List<char>`.

4. *Array.* If T is a type and i is an integer, then `Array<T,i>` denotes the type whose elements are arrays of i elements of type T. For example, `Array<char,10>` denotes character strings of length 10.

5. *Dictionary.* If T and S are types, then `Dictionary<T,S>` denotes a type whose values are finite sets of pairs. Each pair consists of a value of the *key type* T and a value of the *range type* S. The dictionary may not contain two pairs with the same key value. Presumably, the dictionary is implemented in a way that makes it very efficient, given a value t of the key type T, to find the associated value of the range type S.

6. *Structures.* If $T_1, T_2, \ldots, T_n$ are types, and $F_1, F_2, \ldots, F_n$ are names of fields, then

> **Sets, Bags, and Lists**
>
> To understand the distinction between sets, bags, and lists, remember that a set has unordered elements, and only one occurrence of each element. A bag allows more than one occurrence of an element, but the elements and their occurrences are unordered. A list allows more than one occurrence of an element, but the occurrences are ordered. Thus, $\{1,2,1\}$ and $\{2,1,1\}$ are the same bag, but $(1,2,1)$ and $(2,1,1)$ are not the same list.

`Struct N {T1 F1, T2 F2, ..., Tn Fn}`

denotes the type named N whose elements are structures with n fields. The ith field is named F_i and has type T_i. For example, line (10) of Fig. 4.3 showed a structure type named `Addr`, with two fields. Both fields are of type `string` and have names `street` and `city`, respectively.

The first five types — set, bag, list, array, and dictionary — are called *collection types*. There are different rules about which types may be associated with attributes and which with relationships.

- The type of a relationship is either a class type or a (single use of a) collection type constructor applied to a class type.

- The type of an attribute is built starting with an atomic type or types. Class types may also be used, but typically these will be classes that are used as "structures," much as the `Addr` structure was used in Example 4.3. We generally prefer to connect classes with relationships, because relationships are two-way, which makes queries about the database easier to express. In contrast, we can go from an object to its attributes, but not vice-versa. After beginning with atomic or class types, we may then apply the structure and collection type constructors as we wish, as many times as we wish.

Example 4.8: Some of the possible types of attributes are:

1. `integer`.

2. `Struct N {string field1, integer field2}`.

3. `List<real>`.

4. `Array<Struct N {string field1, integer field2}, 10>`.

Example (1) is an atomic type; (2) is a structure of atomic types, (3) a collection of an atomic type, and (4) a collection of structures built from atomic types.

Now, suppose the class names `Movie` and `Star` are available basic types. Then we may construct relationship types such as `Movie` or `Bag<Star>`. However, the following are illegal as relationship types:

1. `Struct N {Movie field1, Star field2}`. Relationship types cannot involve structures.
2. `Set<integer>`. Relationship types cannot involve atomic types.
3. `Set<Array<Star, 10>>`. Relationship types cannot involve two applications of collection types.

□

4.2.9 Exercises for Section 4.2

* **Exercise 4.2.1:** In Exercise 2.1.1 was the informal description of a bank database. Render this design in ODL.

Exercise 4.2.2: Modify your design of Exercise 4.2.1 in the ways enumerated in Exercise 2.1.2. Describe the changes; do not write a complete, new schema.

Exercise 4.2.3: Render the teams-players-fans database of Exercise 2.1.3 in ODL. Why does the complication about sets of team colors, which was mentioned in the original exercise, not present a problem in ODL?

*! **Exercise 4.2.4:** Suppose we wish to keep a genealogy. We shall have one class, `Person`. The information we wish to record about persons includes their name (an attribute) and the following relationships: mother, father, and children. Give an ODL design for the `Person` class. Be sure to indicate the inverses of the relationships that, like `mother`, `father`, and `children`, are also relationships from `Person` to itself. Is the inverse of the `mother` relationship the `children` relationship? Why or why not? Describe each of the relationships and their inverses as sets of pairs.

! **Exercise 4.2.5:** Let us add to the design of Exercise 4.2.4 the attribute `education`. The value of this attribute is intended to be a collection of the degrees obtained by each person, including the name of the degree (e.g., B.S.), the school, and the date. This collection of structs could be a set, bag, list, or array. Describe the consequences of each of these four choices. What information could be gained or lost by making each choice? Is the information lost likely to be important in practice?

Exercise 4.2.6: In Fig. 4.5 is an ODL definition for the classes `Ship` and `TG` (*task group*, a collection of ships). We would like to make some modifications

to this definition. Each modification can be described by mentioning a line or lines to be changed and giving the replacement, or by inserting one or more new lines after one of the numbered lines. Describe the following modifications:

a) The type of the attribute `commander` is changed to be a pair of strings, the first of which is the rank and the second of which is the name.

b) A ship is allowed to be assigned to more than one task group.

c) *Sister ships* are identical ships made from the same plans. We wish to represent, for each ship, the set of its sister ships (other than itself). You may assume that each ship's sister ships are `Ship` objects.

```
1)  class Ship {
2)      attribute string name;
3)      attribute integer yearLaunched;
4)      relationship TG assignedTo inverse TG::unitsOf;
    };

5)  class TG {
6)      attribute real number;
7)      attribute string commander;
8)      relationship Set<Ship> unitsOf
            inverse Ship::assignedTo;
    };
```

Figure 4.5: An ODL description of ships and task groups

*!! **Exercise 4.2.7:** Under what circumstances is a relationship its own inverse? *Hint*: Think about the relationship as a set of pairs, as discussed in Section 4.2.5.

4.3 Additional ODL Concepts

There are a number of other features of ODL that we must learn if we are to express in ODL the things that we can express in the E/R or relational models. In this section, we shall cover:

1. Representing multiway relationships. Notice that all ODL relationships are binary, and we have to go to some lengths to represent 3-way or higher arity relationships that are simple to represent in E/R diagrams or relations.
2. Subclasses and inheritance.

3. Keys, which are optional in ODL.

4. Extents, the set of objects of a given class that exist in a database. These are the ODL equivalent of entity sets or relations, and must not be confused with the class itself, which is a schema.

4.3.1 Multiway Relationships in ODL

ODL supports only binary relationships. There is a trick, which we introduced in Section 2.1.7, to replace a multiway relationship by several binary, many-one relationships. Suppose we have a multiway relationship R among classes or entity sets $C_1, C_2, \ldots, C_n$. We may replace R by a class C and n many-one binary relationships from C to each of the C_i's. Each object of class C may be thought of as a tuple t in the relationship set for R. Object t is related, by the n many-one relationships, to the objects of the classes C_i that participate in the relationship-set tuple t.

Example 4.9: Let us consider how we would represent in ODL the 3-way relationship *Contracts*, whose E/R diagram was given in Fig. 2.7. We may start with the class definitions for `Movie`, `Star`, and `Studio`, the three classes that are related by *Contracts*, that we saw in Fig. 4.3.

We must create a class `Contract` that corresponds to the 3-way relationship *Contracts*. The three many-one relationships from `Contract` to the other three classes we shall call `theMovie`, `theStar`, and `theStudio`. Figure 4.6 shows the definition of the class `Contract`.

```
1)   class Contract {
2)       attribute integer salary;
3)       relationship Movie theMovie
             inverse ... ;
4)       relationship Star theStar
             inverse ... ;
5)       relationship Studio theStudio
             inverse ... ;
     };
```

Figure 4.6: A class `Contract` to represent the 3-way relationship *Contracts*

There is one attribute of the class `Contract`, the salary, since that quantity is associated with the contract itself, not with any of the three participants. Recall that in Fig. 2.7 we made an analogous decision to place the attribute *salary* on the relationship *Contracts*, rather than on one of the participating entity sets. The other properties of `Contract` objects are the three relationships mentioned.

Note that we have not named the inverses of these relationships. We need to modify the declarations of `Movie`, `Star`, and `Studio` to include relationships

from each of these to `Contract`. For instance, the inverse of `theMovie` might be named `contractsFor`. We would then replace line (3) of Fig. 4.6 by

```
3)   relationship Movie theMovie
         inverse Movie::contractsFor;
```

and add to the declaration of `Movie` the statement:

```
relationship Set<Contract> contractsFor
    inverse Contract::theMovie;
```

Notice that in `Movie`, the relationship `contractsFor` gives us a set of contracts, since there may be several contracts associated with one movie. Each contract in the set is essentially a triple consisting of that movie, a star, and a studio, plus the salary that is paid to the star by the studio for acting in that movie. □

4.3.2 Subclasses in ODL

Let us recall the discussion of subclasses in the E/R model from Section 2.1.11. There is a similar capability in ODL to declare one class C to be a subclass of another class D. We follow the name C in its declaration with the keyword `extends` and the name D.

Example 4.10: Recall Example 2.10, where we declared cartoons to be a subclass of movies, with the additional property of a relationship from a cartoon to a set of stars that are its "voices." We can create a subclass `Cartoon` for `Movie` with the ODL declaration:

```
class Cartoon extends Movie {
   relationship Set<Star> voices;
};
```

We have not indicated the name of the inverse of relationship `voices`, although technically we must do so.

A subclass *inherits* all the properties of its superclass. Thus, each cartoon object has attributes `title`, `year`, `length`, and `filmType` inherited from `Movie` (recall Fig. 4.3), and it inherits relationships `stars` and `ownedBy` from `Movie`, in addition to its own relationship `voices`.

Also in that example, we defined a class of murder mysteries with additional attribute `weapon`.

```
class MurderMystery extends Movie {
   attribute string weapon;
};
```

is a suitable declaration of this subclass. Again, all the properties of movies are inherited by `MurderMystery`. □

4.3.3 Multiple Inheritance in ODL

Sometimes, as in the case of a movie like "Roger Rabbit," we need a class that is a subclass of two or more other classes at the same time. In the E/R model, we were able to imagine that "Roger Rabbit" was represented by components in all three of the *Movies*, *Cartoons*, and *Murder-Mysteries* entity sets, which were connected in an isa-hierarchy. However, a principle of object-oriented systems is that objects belong to one and only one class. Thus, to represent movies that are both cartoons and murder mysteries, we need a fourth class for these movies.

The class `CartoonMurderMystery` must inherit properties from both `Cartoon` and `MurderMystery`, as suggested by Fig. 4.7. That is, a `CartoonMurderMystery` object has all the properties of a `Movie` object, plus the relationship `voices` and the attribute `weapon`.

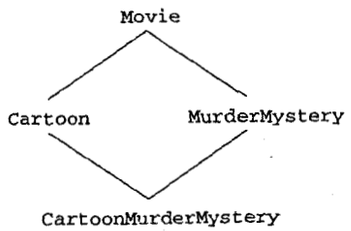

Figure 4.7: Diagram showing multiple inheritance

In ODL, we may follow the keyword `extends` by several classes, separated by colons.[3] Thus, we may declare the fourth class by:

```
class CartoonMurderMystery
    extends MurderMystery : Cartoon;
```

When a class C inherits from several classes, there is the potential for *conflicts* among property names. Two or more of the superclasses of C may have a property of the same name, and the types of these properties may differ. Class `CartoonMurderMystery` did not present such a problem, since the only properties in common between `Cartoon` and `MurderMystery` are the properties of `Movie`, which are the same property in both superclasses of `CartoonMurderMystery`. Here is an example where we are not so lucky.

Example 4.11: Suppose we have subclasses of `Movie` called `Romance` and `Courtroom`. Further suppose that each of these subclasses has an attribute called `ending`. In class `Romance`, attribute `ending` draws its values from the enumeration `{happy, sad}`, while in class `Courtroom`, attribute `ending` draws its values from the enumeration `{guilty, notGuilty}`. If we create a further subclass, `Courtroom-Romance`, that has as superclasses both `Romance` and `Courtroom`, then the type for inherited attribute `ending` in class `Courtroom-Romance` is unclear. □

[3]Technically, the second and subsequent names must be "interfaces," rather than classes. Roughly, an *interface* in ODL is a class definition without an associated set of objects, or "extent." We discuss the distinction further in Section 4.3.4.

The ODL standard does not dictate how such conflicts are to be resolved. Some possible approaches to handling conflicts that arise from multiple inheritance are:

1. Disallow multiple inheritance altogether. This approach is generally regarded as too limiting.
2. Indicate which of the candidate definitions of the property applies to the subclass. For instance, in Example 4.11 we may decide that in a courtroom romance we are more interested in whether the movie has a happy or sad ending than we are in the verdict of the courtroom trial. In this case, we would specify that class `Courtroom-Romance` inherits attribute `ending` from superclass `Romance`, and not from superclass `Courtroom`.
3. Give a new name in the subclass for one of the identically named properties in the superclasses. For instance, in Example 4.11, if `Courtroom-Romance` inherits attribute `ending` from superclass `Romance`, then we may specify that class `Courtroom-Romance` has an additional attribute called `verdict`, which is a renaming of the attribute `ending` inherited from class `Courtroom`.

4.3.4 Extents

When an ODL class is part of the database being defined, we need to distinguish the class definition itself from the set of objects of that class that exist at a given time. The distinction is the same as that between a relation schema and a relation instance, even though both can be referred to by the name of the relation, depending on context. Likewise, in the E/R model we need to distinguish between the definition of an entity set and the set of existing entities of that kind.

In ODL, the distinction is made explicit by giving the class and its *extent*, or set of existing objects, different names. Thus, the class name is a schema for the class, while the extent is the name of the current set of objects of that class. We provide a name for the extent of a class by following the class name by a parenthesized expression consisting of the keyword `extent` and the name chosen for the extent.

Example 4.12: In general, we find it a useful convention to name classes by a singular noun and name the corresponding extent by the same noun in plural. Following this convention, we could call the extent for class `Movie` by the name

Interfaces

ODL provides for the definition of *interfaces*, which are essentially class definitions with no associated extent (and therefore, with no associated objects). We first mentioned interfaces in Section 4.3.3, where we pointed out that they could support inheritance by one class from several classes. Interfaces also are useful if we have several classes that have different extents, but the same properties; the situation is analogous to several relations that have the same schema but different sets of tuples.

If we define an interface I, we can then define several classes that inherit their properties from I. Each of those classes has a distinct extent, so we can maintain in our database several sets of objects that have the same type, yet belong to distinct classes.

`Movies`. To declare this name for the extent, we would begin the declaration of class `Movie` by:

```
class Movie (extent Movies) {
    attribute string title;
        ...
```

As we shall see when we study the query language OQL that is designed for querying ODL data, we refer to the extent `Movies`, not to the class `Movie`, when we want to examine the movies currently stored in our database. Remember that the choice of a name for the extent of a class is entirely arbitrary, although we shall follow the "make it plural" convention in this book. □

4.3.5 Declaring Keys in ODL

ODL differs from the other models studied so far in that the declaration and use of keys is optional. That is, in the E/R model, entity sets need keys to distinguish members of the entity set from one another. In the relational model, where relations are sets, all attributes together form a key unless some proper subset of the attributes for a given relation can serve as a key. Either way, there must be at least one key for a relation.

However, objects have a unique object identity, as we discussed in Section 4.1.3. Consequently, in ODL, the declaration of a key or keys is optional. It is entirely appropriate for there to be several objects of a class that are indistinguishable by any properties we can observe; the system still keeps them distinct by their internal object identity.

In ODL we may declare one or more attributes to be a key for a class by using the keyword `key` or `keys` (it doesn't matter which) followed by the attribute

or attributes forming keys. If there is more than one attribute in a key, the list of attributes must be surrounded by parentheses. The key declaration itself appears, along with the extent declaration, inside parentheses that may follow the name of the class itself in the first line of its declaration.

Example 4.13: To declare that the set of two attributes `title` and `year` form a key for class `Movie`, we could begin its declaration:

```
class Movie
    (extent Movies key (title, year))
{
    attribute string title;
        ...
```

We could have used `keys` in place of `key`, even though only one key is declared. Similarly, if `name` is a key for class `Star`, then we could begin its declaration:

```
class Star
    (extent Stars key name)
{
    attribute string name;
        ...
```

□

It is possible that several sets of attributes are keys. If so, then following the word `key(s)` we may place several keys separated by commas. As usual, a key that consists of more than one attribute must have parentheses around the list of its attributes, so we can disambiguate a key of several attributes from several keys of one attribute each.

Example 4.14: As an example of a situation where it is appropriate to have more than one key, consider a class `Employee`, whose complete set of attributes and relationships we shall not describe here. However, suppose that two of its attributes are `empID`, the employee ID, and `ssNo`, the Social Security number. Then we can declare each of these attributes to be a key by itself with

```
class Employee
   (extent Employees key empID, ssNo)
      ...
```

Because there are no parentheses around the list of attributes, ODL interprets the above as saying that each of the two attributes is a key by itself. If we put parentheses around the list `(empID, ssNo)`, then ODL would interpret the two attributes together as forming one key. That is, the implication of writing

```
class Employee
   (extent Employees key (empID, ssNo))
      ...
```

is that no two employees could have both the same employee ID and the same Social Security number, although two employees might agree on one of these attributes. □

The ODL standard also allows properties other than attributes to appear in keys. There is no fundamental problem with a method or relationship being declared a key or part of a key, since keys are advisory statements that the DBMS can take advantage of or not, as it wishes. For instance, one could declare a method to be a key, meaning that on distinct objects of the class the method is guaranteed to return distinct values.

When we allow many-one relationships to appear in key declarations, we can get an effect similar to that of weak entity sets in the E/R model. We can declare that the object O_1 referred to by an object O_2 on the "many" side of the relationship, perhaps together with other properties of O_2 that are included in the key, is unique for different objects O_2. However, we should remember that there is no requirement that classes have keys; we are never obliged to handle, in some special way, classes that lack attributes of their own to form a key, as we did for weak entity sets.

Example 4.15: Let us review the example of a weak entity set *Crews* in Fig. 2.20. Recall that we hypothesized that crews were identified by their number, and the studio for which they worked, although two studios might have crews with the same number. We might declare the class `Crew` as in Fig. 4.8. Note that we need to modify the declaration of `Studio` to include the relationship `crewsOf` that is an inverse to the relationship `unitOf` in `Crew`; we omit this change.

```
class Crew
    (extent Crews key (number, unitOf))
{
    attribute integer number;
    relationship Studio unitOf
        inverse Studio::crewsOf;
}
```

Figure 4.8: A ODL declaration for crews

What this key declaration asserts is that there cannot be two crews that both have the same value for the `number` attribute and are related to the same studio by `unitOf`. Notice how this assertion resembles the implication of the E/R diagram in Fig. 2.20, which is that the number of a crew and the name of the related studio (i.e., the key for studios) uniquely determine a crew entity. □

4.3.6 Exercises for Section 4.3

Exercise 4.3.1: Add suitable extents and keys to your ODL schema from Exercise 4.2.1.

Exercise 4.3.2: Add suitable extents and keys to your ODL schema from Exercise 4.2.3.

! **Exercise 4.3.3:** Suppose we wish to modify the ODL declarations of Exercise 4.2.4, where we had a class of people with relationships `mother`, `father`, and `children`, to include certain subclasses of people: (1) Males (2) Females (3) People who are parents. In addition, we want the relationships `mother`, `father`, and `children` to run between the smallest classes for which all possible instances of the relationship appear. You may therefore wish to define other subclasses as well. Write these declarations, including multiple inheritances when appropriate.

Exercise 4.3.4: Is there a suitable key for the class `Contract` declared in Fig. 4.6? If so, what is it?

Exercise 4.3.5: In Exercise 2.4.4 we saw two examples of situations where weak entity sets were essential. Render these databases in ODL, including declarations for extents and suitable keys.

Exercise 4.3.6: Give an ODL design for the registrar's database described in Exercise 2.1.9.

4.4 From ODL Designs to Relational Designs

While the E/R model is intended to be converted into a model such as the relational model when we implement the design as an actual database, ODL was originally intended to be used as the specification language for real, object-oriented DBMS's. However ODL, like all object-oriented design systems, can also be used for preliminary design and converted to relations prior to implementation. In this section we shall consider how to convert ODL designs into relational designs. The process is similar in many ways to what we introduced in Section 3.2 for converting E/R diagrams to relational database schemas. Yet some new problems arise for ODL, including:

1. Entity sets must have keys, but there is no such guarantee for ODL classes. Therefore, in some situations we must invent a new attribute to serve as a key when we construct a relation for the class.
2. While we have required E/R attributes and relational attributes to be atomic, there is no such constraint for ODL attributes. The conversion of attributes that have collection types to relations is tricky and ad-hoc, often resulting in unnormalized relations that must be redesigned by the techniques of Section 3.6.

3. ODL allows us to specify methods as part of a design, but there is no simple way to convert methods directly into a relational schema. We shall visit the issue of methods in relational schemas in Section 4.5.5 and again in Chapter 9 covering the SQL-99 standard. For now, let us assume that any ODL design we wish to convert into a relational design does not include methods.

4.4.1 From ODL Attributes to Relational Attributes

As a starting point, let us assume that our goal is to have one relation for each class and for that relation to have one attribute for each property. We shall see many ways in which this approach must be modified, but for the moment, let us consider the simplest possible case, where we can indeed convert classes to relations and properties to attributes. The restrictions we assume are:

1. All properties of the class are attributes (not relationships or methods).
2. The types of the attributes are atomic (not structures or sets).

Example 4.16: Figure 4.9 is an example of such a class. There are four attributes and no other properties. These attributes each have an atomic type; `title` is a string, `year` and `length` are integers, and `filmType` is an enumeration of two values.

```
class Movie (extent Movies) {
   attribute string title;
   attribute integer year;
   attribute integer length;
   attribute enum Film {color,blackAndWhite} filmType;
};
```

Figure 4.9: Attributes of the class `Movie`

We create a relation with the same name as the extent of the class, `Movies` in this case. The relation has four attributes, one for each attribute of the class. The names of the relational attributes can be the same as the names of the corresponding class attributes. Thus, the schema for this relation is

```
Movies(title, year, length, filmType)
```

For each object in the extent `Movies`, there is one tuple in the relation `Movies`. This tuple has a component for each of the four attributes, and the value of each component is the same as the value of the corresponding attribute of the object. □

4.4.2 Nonatomic Attributes in Classes

Unfortunately, even when a class' properties are all attributes we may have some difficulty converting the class to a relation. The reason is that attributes in ODL can have complex types such as structures, sets, bags, or lists. On the other hand, a fundamental principle of the relational model is that a relation's attributes have an atomic type, such as numbers and strings. Thus, we must find some way of representing nonatomic attribute types as relations.

Record structures whose fields are themselves atomic are the easiest to handle. We simply expand the structure definition, making one attribute of the relation for each field of the structure. The only possible problem is that two structures could have fields of the same name, in which case we have to invent new attribute names to distinguish them in the relation.

```
class Star (extent Stars) {
   attribute string name;
   attribute Struct Addr
       {string street, string city} address;
};
```

Figure 4.10: Class with a structured attribute

Example 4.17: In Fig. 4.10 is a declaration for class `Star`, with only attributes as properties. The attribute `name` is atomic, but attribute `address` is a structure with two fields, `street` and `city`. Thus, we can represent this class by a relation with three attributes. The first attribute, `name`, corresponds to the ODL attribute of the same name. The second and third attributes we shall call `street` and `city`; they correspond to the two fields of the `address` structure and together represent an address. Thus, the schema for our relation is

```
Stars(name, street, city)
```

Figure 4.11 shows some typical tuples of this relation. □

name	*street*	*city*
Carrie Fisher	123 Maple St.	Hollywood
Mark Hamill	456 Oak Rd.	Brentwood
Harrison Ford	789 Palm Dr.	Beverly Hills

Figure 4.11: A relation representing stars

4.4.3 Representing Set-Valued Attributes

However, record structures are not the most complex kind of attribute that can appear in ODL class definitions. Values can also be built using type constructors `Set`, `Bag`, `List`, `Array`, and `Dictionary` from Section 4.2.8. Each presents its own problems when migrating to the relational model. We shall only discuss the `Set` constructor, which is the most common, in detail.

One approach to representing a set of values for an attribute A is to make one tuple for each value. That tuple includes the appropriate values for all the other attributes besides A. Let us first see an example where this approach works well, and then we shall see a pitfall.

```
class Star (extent Stars) {
   attribute string name;
   attribute Set<
          Struct Addr {string street, string city}
       > address;
};
```

Figure 4.12: Stars with a set of addresses

Example 4.18: Suppose that class `Star` were defined so that for each star we could record a set of addresses, as in Fig. 4.12. Suppose next that Carrie Fisher also has a beach home, but the other two stars mentioned in Fig. 4.11 each have only one home. Then we may create two tuples with `name` attribute equal to `"Carrie Fisher"`, as shown in Fig. 4.13. Other tuples remain as they were in Fig. 4.11. □

name	*street*	*city*
Carrie Fisher	123 Maple St.	Hollywood
Carrie Fisher	5 Locust Ln.	Malibu
Mark Hamill	456 Oak Rd.	Brentwood
Harrison Ford	789 Palm Dr.	Beverly Hills

Figure 4.13: Allowing a set of addresses

Unfortunately, this technique of replacing objects with one or more set-valued attributes by collections of tuples, one for each combination of values for these attributes, can lead to unnormalized relations, of the type discussed in Section 3.6. In fact, even one set-valued attribute can lead to a BCNF violation, as the next example shows.

Atomic Values: Bug or Feature?

It seems that the relational model puts obstacles in our way, while ODL is more flexible in allowing structured values as properties. One might be tempted to dismiss the relational model altogether or regard it as a primitive concept that has been superseded by more elegant "object-oriented" approaches such as ODL. However, the reality is that database systems based on the relational model are dominant in the marketplace. One of the reasons is that the simplicity of the model makes possible powerful programming languages for querying databases, especially SQL (see Chapter 6), the standard language used in most of today's database systems.

```
class Star (extent Stars) {
   attribute string name;
   attribute Set<
          Struct Addr {string street, string city}
       > address;
   attribute Date birthdate;
};
```

Figure 4.14: Stars with a set of addresses and a birthdate

Example 4.19: Suppose that we add `birthdate` as an attribute in the definition of the `Star` class; that is, we use the definition shown in Fig. 4.14. We have added to Fig. 4.12 the attribute `birthdate` of type `Date`, which is one of ODL's atomic types. The `birthdate` attribute can be an attribute of the `Stars` relation, whose schema now becomes:

```
Stars(name, street, city, birthdate)
```

Let us make another change to the data of Fig. 4.13. Since a set of addresses can be empty, let us assume that Harrison Ford has no address in the database. Then the revised relation is shown in Fig. 4.15. Two bad things have happened:

1. Carrie Fisher's birthdate has been repeated in each tuple, causing redundancy. Note that her name is also repeated, but that repetition is not true redundancy, because without the name appearing in each tuple we could not know that both addresses were associated with Carrie Fisher.

2. Because Harrison Ford has an empty set of addresses, we have lost all information about him. This situation is an example of a deletion anomaly that we discussed in Section 3.6.1.

name	*street*	*city*	*birthdate*
Carrie Fisher	123 Maple St.	Hollywood	9/9/99
Carrie Fisher	5 Locust Ln.	Malibu	9/9/99
Mark Hamill	456 Oak Rd.	Brentwood	8/8/88

Figure 4.15: Adding birthdates

Although `name` is a key for the class `Star`, our need to have several tuples for one star to represent all their addresses means that `name` is *not* a key for the relation `Stars`. In fact, the key for that relation is {`name, street, city`}. Thus, the functional dependency

$$\texttt{name} \rightarrow \texttt{birthdate}$$

is a BCNF violation. This fact explains why the anomalies mentioned above are able to occur. □

There are several options regarding how to handle set-valued attributes that appear in a class declaration along with other attributes, set-valued or not. First, we may simply place all attributes, set-valued or not, in the schema for the relation, then use the normalization techniques of Sections 3.6 and 3.7 to eliminate the resulting BCNF and 4NF violations. Notice that a set-valued attribute in conjunction with a single-valued attribute leads to a BNCF violation, as in Example 4.19. Two set-valued attributes in the same class declaration will lead to a 4NF violation.

The second approach is to separate out each set-valued attribute as if it were a many-many relationship between the objects of the class and the values that appear in the sets. We shall discuss this approach for relationships in Section 4.4.5.

4.4.4 Representing Other Type Constructors

Besides record structures and sets, an ODL class definition could use `Bag`, `List`, `Array`, or `Dictionary` to construct values. To represent a bag (multiset), in which a single object can be a member of the bag n times, we cannot simply introduce into a relation n identical tuples.[4] Instead, we could add to the relation schema another attribute `count` representing the number of times that each element is a member of the bag. For instance, suppose that `address` in Fig. 4.12 were a bag instead of a set. We could say that 123 Maple St., Hollywood is Carrie Fisher's address twice and 5 Locust Ln., Malibu is her address 3 times (whatever that may mean) by

name	*street*	*city*	*count*
Carrie Fisher	123 Maple St.	Hollywood	2
Carrie Fisher	5 Locust Ln.	Malibu	3

A list of addresses could be represented by a new attribute `position`, indicating the position in the list. For instance, we could show Carrie Fisher's addresses as a list, with Hollywood first, by:

name	*street*	*city*	*position*
Carrie Fisher	123 Maple St.	Hollywood	1
Carrie Fisher	5 Locust Ln.	Malibu	2

A fixed-length array of addresses could be represented by attributes for each position in the array. For instance, if `address` were to be an array of two street-city structures, we could represent `Star` objects as:

name	*street1*	*city1*	*street2*	*city2*
Carrie Fisher	123 Maple St.	Hollywood	5 Locust Ln.	Malibu

Finally, a dictionary could be represented as a set, but with attributes for both the key-value and range-value components of the pairs that are members of the dictionary. For instance, suppose that instead of star's addresses, we really wanted to keep, for each star, a dictionary giving the mortgage holder for each of their homes. Then the dictionary would have address as the key value and bank name as the range value. A hypothetical rendering of the Carrie-Fisher object with a dictionary attribute is:

name	*street*	*city*	*mortgage-holder*
Carrie Fisher	123 Maple St.	Hollywood	Bank of Burbank
Carrie Fisher	5 Locust Ln.	Malibu	Torrance Trust

Of course attribute types in ODL may involve more than one type constructor. If a type is any collection type besides dictionary applied to a structure (e.g., a set of structs), then we may apply the techniques from Sections 4.4.3 or 4.4.4 as if the struct were an atomic value, and then replace the single attribute representing the atomic value by several attributes, one for each field of the struct. This strategy was used in the examples above, where the address is a struct. The case of a dictionary applied to structs is similar and left as an exercise.

There are many reasons to limit the complexity of attribute types to an optional struct followed by an optional collection type. We mentioned in Section 2.1.1 that some versions of the E/R model allow exactly this much generality in the types of attributes, although we restricted ourselves to atomic

[4] To be precise, we cannot introduce identical tuples into relations of the abstract relational model described in Chapter 3. However, SQL-based relational DBMS's *do* allow duplicate tuples; i.e., relations are bags rather than sets in SQL. See Sections 5.3 and 6.4. If queries are likely to ask for tuple counts, we advise using a scheme such as that described here, even if your DBMS allows duplicate tuples.

attributes in the E/R model. We recommend that, if you are going to use an ODL design for the purpose of eventual translation to a relational database schema, you similarly limit yourself. We take up in the exercises some options for dealing with more complex types as attributes.

4.4.5 Representing ODL Relationships

Usually, an ODL class definition will contain relationships to other ODL classes. As in the E/R model, we can create for each relationship a new relation that connects the keys of the two related classes. However, in ODL, relationships come in inverse pairs, and we must create only one relation for each pair.

```
class Movie
    (extent Movies key(title, year))
{
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color,blackAndWhite} filmType;
    relationship Set<Star> stars
                inverse Star::starredIn;
    relationship Studio ownedBy
                inverse Studio::owns;
};

class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                inverse Movie::ownedBy;
};
```

Figure 4.16: The complete definition of the Movie and Studio classes

Example 4.20: Consider the declarations of the classes Movie and Studio, which we repeat in Fig. 4.16. We see that title and year form the key for Movie and name is a key for class Studio. We may create a relation for the pair of relationships owns and ownedBy. The relation needs a name, which can be arbitrary; we shall pick StudioOf as the name. The schema for StudioOf has attributes for the key of Movie, that is, title and year, and an attribute that we shall call studioName for the key of Studio. This relation schema is thus:


```
StudioOf(title, year, studioName)
```

Some typical tuples that would be in this relation are:

title	*year*	*studioName*
Star Wars	1977	Fox
Mighty Ducks	1991	Disney
Wayne's World	1992	Paramount

□

When a relationship is many-one, we have an option to combine it with the relation that is constructed for the class on the "many" side. Doing so has the effect of combining two relations that have a common key, as we discussed in Section 3.2.3. It therefore does not cause a BCNF violation and is a legitimate and commonly followed option.

Example 4.21: Rather than creating a relation StudioOf for relationship pair owns and ownedBy, as we did in Example 4.20, we may instead modify our relation schema for relation Movies to include an attribute, say studioName, to represent the key of Studio. If we do, the schema for Movies becomes

```
Movies(title, year, length, filmType, studioName)
```

and some typical tuples for this relation are:

title	*year*	*length*	*filmType*	*studioName*
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

Note that title and year, the key for the Movie class, is also a key for relation Movies, since each movie has a unique length, film type, and owning studio. □

We should remember that it is possible but unwise to treat many-many relationships as we did many-one relationships in Example 4.21. In fact, Example 3.6 in Section 3.2.3 was based on what happens if we try to combine the many-many stars relationship between movies and their stars with the other information in the relation Movies to get a relation with schema:

```
Movies(title, year, length, filmType, studioName, starName)
```

There is a resulting BCNF violation, since {title, year, starName} is the key, yet attributes length, filmType, and studioName each are functionally determined by only title and year.

Likewise, if we do combine a many-one relationship with the relation for a class, it must be the class of the "many." For instance, combining owns and its inverse ownedBy with relation Studios will lead to a BCNF violation (see Exercise 4.4.4).

4.4.6 What If There Is No Key?

Since keys are optional in ODL, we may face a situation where the attributes available to us cannot serve to represent objects of a class C uniquely. That situation can be a problem if the class C participates in one or more relationships.

We recommend creating a new attribute or "certificate" that can serve as an identifier for objects of class C in relational designs, much as the hidden object-ID serves to identify those objects in an object-oriented system. The certificate becomes an additional attribute of the relation for the class C, as well as representing objects of class C in each of the relations that come from relationships involving class C. Notice that in practice, many important classes are represented by such certificates: university ID's for students, driver's-license numbers for drivers, and so on.

Example 4.22: Suppose we accept that names are not a reliable key for movie stars, and we decide instead to adopt a "certificate number" to be assigned to each star as a way of identifying them uniquely. Then the `Stars` relation would have schema:

```
Stars(cert#, name, street, city, birthdate)
```

If we wish to represent the many-many relationship between movies and their stars by a relation `StarsIn`, we can use the `title` and `year` attributes from `Movie` and the certificate to represent stars, giving us a relation with schema:

```
StarsIn(title, year, cert#)
```

□

4.4.7 Exercises for Section 4.4

Exercise 4.4.1: Convert your ODL designs from the following exercises to relational database schemas.

* a) Exercise 4.2.1.

b) Exercise 4.2.2 (include all four of the modifications specified by that exercise).

c) Exercise 4.2.3.

* d) Exercise 4.2.4.

e) Exercise 4.2.5.

Exercise 4.4.2: Convert the ODL description of Fig. 4.5 to a relational database schema. How does each of the three modifications of Exercise 4.2.6 affect your relational schema?

! **Exercise 4.4.3:** Consider an attribute of type dictionary with key and range types both structs of atomic types. Show how to convert a class with an attribute of this type to a relation.

* **Exercise 4.4.4:** We claimed that if you combine the relation for class `Studio`, as defined in Fig. 4.16, with the relation for the relationship pair `owns` and `ownedBy`, then there is a BCNF violation. Do the combination and show that there is, in fact, a BCNF violation.

Exercise 4.4.5: We mentioned that when attributes are of a type more complex than a collection of structs, it becomes tricky to convert them to relations; in particular, it becomes necessary to create some intermediate concepts and relations for them. The following sequence of questions will examine increasingly more complex types and how to represent them as relations.

* a) A *card* can be represented as a struct with fields `rank` (2, 3, ..., 10, Jack, Queen, King, and Ace) and `suit` (Clubs, Diamonds, Hearts, and Spades). Give a suitable definition of a structured type `Card`. This definition should be independent of any class declarations but available to them all.

* b) A *hand* is a set of cards. The number of cards may vary. Give a declaration of a class `Hand` whose objects are hands. That is, this class declaration has an attribute `theHand`, whose type is a hand.

*! c) Convert your class declaration `Hand` from (b) to a relation schema.

d) A *poker hand* is a set of five cards. Repeat (b) and (c) for poker hands.

*! e) A *deal* is a set of pairs, each pair consisting of the name of a player and a hand for that player. Declare a class `Deal`, whose objects are deals. That is, this class declaration has an attribute `theDeal`, whose type is a deal.

f) Repeat (e), but restrict hands of a deal to be hands of exactly five cards.

g) Repeat (e), using a dictionary for a deal. You may assume the names of players in a deal are unique.

*!! h) Convert your class declaration from (e) to a relational database schema.

*! i) Suppose we defined deals to be sets of sets of cards, with no player associated with each hand (set of cards). It is proposed that we represent such deals by a relation schema

```
Deals(dealID, card)
```

meaning that the card was a member of one of the hands in the deal with the given ID. What, if anything, is wrong with this representation? How would you fix the problem?

Exercise 4.4.6: Suppose we have a class C defined by

```
class C (key a) {
    attribute string a;
    attribute T b;
}
```

where T is some type. Give the relation schema for the relation derived from C and indicate its key attributes if T is:

a) `Set<Struct S {string f, string g}>`

*! b) `Bag<Struct S {string f, string g}>`

! c) `List<Struct S {string f, string g}>`

! d) `Dictionary<Struct K {string f, string g}, Struct R {string i, string j}>`

4.5 The Object-Relational Model

The relational model and the object-oriented model typified by ODL are two important points in a spectrum of options that could underlie a DBMS. For an extended period, the relational model was dominant in the commercial DBMS world. Object-oriented DBMS's made limited inroads during the 1990's, but have since died off. Instead of a migration from relational to object-oriented systems, as was widely predicted around 1990, the vendors of relational systems have moved to incorporate many of the ideas found in ODL or other object-oriented-database proposals. As a result, many DBMS products that used to be called "relational" are now called "object-relational."

In Chapter9 we shall meet the new SQL standard for object-relational databases. In this chapter, we cover the topic more abstractly. We introduce the concept of object-relations in Section 4.5.1, then discuss one of its earliest embodiments — nested relations — in Section 4.5.2. ODL-like references for object-relations are discussed in Section 4.5.3, and in Section 4.5.4 we compare the object-relational model against the pure object-oriented approach.

4.5.1 From Relations to Object-Relations

While the relation remains the fundamental concept, the relational model has been extended to the *object-relational model* by incorporation of features such as:

1. *Structured types for attributes.* Instead of allowing only atomic types for attributes, object-relational systems support a type system like ODL's: types built from atomic types and type constructors for structs, sets, and bags, for instance. Especially important is a type that is a set[5] of structs, which is essentially a relation. That is, a value of one component of a tuple can be an entire relation.

2. *Methods.* Special operations can be defined for, and applied to, values of a user-defined type. While we haven't yet addressed the question of how values or tuples are manipulated in the relational or object-oriented models, we shall find few surprises when we take up the subject beginning in Chapter 5. For example, values of numeric type are operated on by arithmetic operators such as addition or less-than. However, in the object-relational model, we have the option to define specialized operations for a type, such as those discussed in Example 4.7 on ODL methods for the `Movie` class.

3. *Identifiers for tuples.* In object-relational systems, tuples play the role of objects. It therefore becomes useful in some situations for each tuple to have a unique ID that distinguishes it from other tuples, even from tuples that have the same values in all components. This ID, like the object-identifier assumed in ODL, is generally invisible to the user, although there are even some circumstances where users can see the identifier for a tuple in an object-relational system.

4. *References.* While the pure relational model has no notion of references or pointers to tuples, object-relational systems can use these references in various ways.

In the next sections, we shall elaborate and illustrate each of these additional capabilities of object-relational systems.

4.5.2 Nested Relations

Relations extended by point (1) above are often called "nested relations." In the *nested-relational model*, we allow attributes of relations to have a type that is not atomic; in particular, a type can be a relation schema. As a result, there is a convenient, recursive definition of the types of attributes and the types (schemas) of relations:

BASIS: An atomic type (integer, real, string, etc.) can be the type of an attribute.

INDUCTION: A relation's type can be any *schema* consisting of names for one or more attributes, and any legal type for each attribute. In addition, a schema can also be the type of any attribute.

In our discussion of the relational model, we did not specify the particular atomic type associated with each attribute, because the distinctions among

[5] Strictly speaking, a bag rather than a set, since commercial relational DBMS's prefer to support relations with duplicate tuples, i.e. bags, rather than sets.

integers, reals, strings, and so on had little to do with the issues discussed, such as functional dependencies and normalization. We shall continue to avoid this distinction, but when describing the schema of a nested relation, we must indicate which attributes have relation schemas as types. To do so, we shall treat these attributes as if they were the names of relations and follow them by a parenthesized list of their attributes. Those attributes, in turn, may have associated lists of attributes, down for as many levels as we wish.

Example 4.23: Let us design a nested relation schema for stars that incorporates within the relation an attribute `movies`, which will be a relation representing all the movies in which the star has appeared. The relation schema for attribute `movies` will include the title, year, and length of the movie. The relation schema for the relation `Stars` will include the name, address, and birthdate, as well as the information found in `movies`. Additionally, the `address` attribute will have a relation type with attributes `street` and `city`. We can record in this relation several addresses for the star. The schema for `Stars` can be written:

```
Stars(name, address(street, city), birthdate,
   movies(title, year, length))
```

An example of a possible relation for nested relation `Stars` is shown in Fig. 4.17. We see in this relation two tuples, one for Carrie Fisher and one for Mark Hamill. The values of components are abbreviated to conserve space, and the dashed lines separating tuples are only for convenience and have no notational significance.

name	*address*	*birthdate*	*movies*
Fisher	*street* | *city* Maple | H'wood Locust | Malibu	9/9/99	*title* | *year* | *length* Star Wars | 1977 | 124 Empire | 1980 | 127 Return | 1983 | 133
Hamill	*street* | *city* Oak | B'wood	8/8/88	*title* | *year* | *length* Star Wars | 1977 | 124 Empire | 1980 | 127 Return | 1983 | 133

Figure 4.17: A nested relation for stars and their movies

In the Carrie Fisher tuple, we see her name, an atomic value, followed by a relation for the value of the address component. That relation has two

attributes, `street` and `city`, and there are two tuples, corresponding to her two houses. Next comes the birthdate, another atomic value. Finally, there is a component for the `movies` attribute; this attribute has a relation schema as its type, with components for the title, year, and length of a movie. The relation for the `movies` component of the Carrie Fisher tuple has tuples for her three best-known movies.

The second tuple, for Mark Hamill, has the same components. His relation for `address` has only one tuple, because in our imaginary data, he has only one house. His relation for `movies` looks just like Carrie Fisher's because their best-known movies happen, by coincidence, to be the same. Note that these two relations are two different tuple-components. These components happen to be identical, just like two components that happened to have the same integer value, e.g., 124. □

4.5.3 References

The fact that movies like *Star Wars* will appear in several relations that are values of the `movies` attribute in the nested relation `Stars` is a cause of redundancy. In effect, the schema of Example 4.23 has the nested-relation analog of not being in BCNF. However, decomposing this `Stars` relation will not eliminate the redundancy. Rather, we need to arrange that among all the tuples of all the `movies` relations, a movie appears only once.

To cure the problem, object-relations need the ability for one tuple t to refer to another tuple s, rather than incorporating s directly in t. We thus add to our model an additional inductive rule: the type of an attribute can also be a reference to a tuple with a given schema.

If an attribute A has a type that is a reference to a single tuple with a relation schema named R, we show the attribute A in a schema as $A(*R)$. Notice that this situation is analogous to an ODL relationship A whose type is R; i.e., it connects to a single object of type R. Similarly, if an attribute A has a type that is a set of references to tuples of schema R, then A will be shown in a schema as $A(\{*R\})$. This situation resembles an ODL relationship A that has type `Set<`R`>`.

Example 4.24: An appropriate way to fix the redundancy in Fig. 4.17 is to use two relations, one for stars and one for movies. The relation `Movies` will be an ordinary relation with the same schema as the attribute `movies` in Example 4.23. The relation `Stars` will have a schema similar to the nested relation `Stars` of that example, but the movies attribute will have a type that is a set of references to `Movies` tuples. The schemas of the two relations are thus:

```
Movies(title, year, length)
Stars(name, address(street, city), birthdate,
    movies({*Movies}))
```

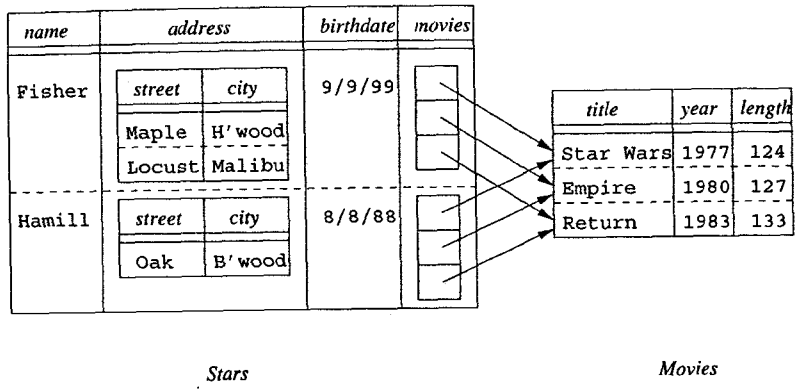

Figure 4.18: Sets of references as the value of an attribute

The data of Fig. 4.17, converted to this new schema, is shown in Fig. 4.18. Notice that, because each movie has only one tuple, although it can have many references, we have eliminated the redundancy inherent in the schema of Example 4.23. □

4.5.4 Object-Oriented Versus Object-Relational

The object-oriented data model, as typified by ODL, and the object-relational model discussed here, are remarkably similar. Some of the salient points of comparison follow.

Objects and Tuples

An object's value is really a struct with components for its attributes and relationships. It is not specified in the ODL standard how relationships are to be represented, but we may assume that an object is connected to related objects by some collection of pointers. A tuple is likewise a struct, but in the conventional relational model, it has components for only the attributes. Relationships would be represented by tuples in another relation, as suggested in Section 3.2.2. However the object-relational model, by allowing sets of references to be a component of tuples, also allows relationships to be incorporated directly into the tuples that represent an "object" or entity.

Extents and Relations

ODL treats all objects in a class as living in an "extent" for that class. The object-relational model allows several different relations with identical schemas, so it might appear that there is more opportunity in the object-relational model to distinguish members of the same class. However, ODL allows the definition of

interfaces, which are essentially class declarations without an extent (see the box on "Interfaces" in Section 4.3.4). Then, ODL allows you to define any number of classes that inherit this interface, while each class has a distinct extent. In that manner, ODL offers the same opportunity the object-relational approach when it comes to sharing the same declaration among several collections.

Methods

We did not discuss the use of methods as part of an object-relational schema. However, in practice, the SQL-99 standard and all implementations of object-relational ideas allow the same ability as ODL to declare and define methods associated with any class.

Type Systems

The type systems of the object-oriented and object-relational models are quite similar. Each is based on atomic types and construction of new types by struct- and collection-type-constructors. The selection of collection types may vary, but all variants include at least sets and bags. Moreover, the set (or bag) of structs type plays a special role in both models. It is the type of classes in ODL, and the type of relations in the object-relational model.

References and Object-ID's

A pure object-oriented model uses object-ID's that are completely hidden from the user, and thus cannot be seen or queried. The object-relational model allows references to be part of a type, and thus it is possible under some circumstances for the user to see their values and even remember them for future use. You may regard this situation as anything from a serious bug to a stroke of genius, depending on your point of view, but in practice it appears to make little difference.

Backwards Compatibility

With little difference in essential features of the two models, it is interesting to consider why object-relational systems have dominated the pure object-oriented systems in the marketplace. The reason, we believe, is that there was, by the time object-oriented systems were seriously proposed, an enormous number of installations running a relational database system. As relational DBMS's evolved into object-relational DBMS's, the vendors were careful to maintain backwards compatibility. That is, newer versions of the system would still run the old code and accept the same schemas, should the user not care to adopt any of the object-oriented features. On the other hand, migration to a pure object-oriented DBMS would require the installations to rewrite and reorganize extensively. Thus, whatever competitive advantage existed was not enough to convert many databases to a pure object-oriented DBMS.

4.5.5 From ODL Designs to Object-Relational Designs

In Section 4.4 we learned how to convert designs in ODL into schemas of the relational model. Difficulties arose primarily because of the richer modeling constructs of ODL: nonatomic attribute types, relationships, and methods. Some — but not all — of these difficulties are alleviated when we translate an ODL design into an object-relational design. Depending on the specific object-relational model used (we shall consider the concrete SQL-99 model in Chapter 9), we may be able to convert most of the nonatomic types of ODL directly into a corresponding object-relational type; structs, sets, bags, lists, and arrays all fall into this category.

If a type in an ODL design is not available in our object-relational model, we can fall back on the techniques from Sections 4.4.2 through 4.4.4. The representation of relationships in an object-relational model is essentially the same as in the relational model (see Section 4.4.5), although we may prefer to use references in place of keys. Finally, although we were not able to translate ODL designs with methods into the pure relational model, most object-relational models include methods, so this restriction can be lifted.

4.5.6 Exercises for Section 4.5

Exercise 4.5.1: Using the notation developed for nested relations and relations with references, give one or more relation schemas that represent the following information. In each case, you may exercise some discretion regarding what attributes of a relation are included, but try to keep close to the attributes found in our running movie example. Also, indicate whether your schemas exhibit redundancy, and if so, what could be done to avoid it.

* a) Movies, with the usual attributes plus all their stars and the usual information about the stars.

*! b) Studios, all the movies made by that studio, and all the stars of each movie, including all the usual attributes of studios, movies, and stars.

c) Movies with their studio, their stars, and all the usual attributes of these.

* **Exercise 4.5.2:** Represent the banking information of Exercise 2.1.1 in the object-relational model developed in this section. Make sure that it is easy, given the tuple for a customer, to find their account(s) and *also* easy, given the tuple for an account to find the customer(s) that hold that account. Also, try to avoid redundancy.

Exercise 4.5.3: If the data of Exercise 4.5.2 were modified so that an account could be held by only one customer [as in Exercise 2.1.2(a)], how could your answer to Exercise 4.5.2 be simplified?

Exercise 4.5.4: Render the players, teams, and fans of Exercise 2.1.3 in the object-relational model.

! **Exercise 4.5.5:** Render the genealogy of Exercise 2.1.6 in the object-relational model.

4.6 Semistructured Data

The *semistructured-data* model plays a special role in database systems:

1. It serves as a model suitable for *integration* of databases, that is, for describing the data contained in two or more databases that contain similar data with different schemas.

2. It serves as a document model in notations such as XML, to be taken up in Section 4.7, that are being used to share information on the Web.

In this section, we shall introduce the basic ideas behind "semistructured data" and how it can represent information more flexibly than the other models we have met previously.

4.6.1 Motivation for the Semistructured-Data Model

Let us begin by recalling the E/R model, and its two fundamental kinds of data — the entity set and the relationship. Remember also that the relational model has only one kind of data — the relation, yet we saw in Section 3.2 how both entity sets and relationships could be represented by relations. There is an advantage to having two concepts: we could tailor an E/R design to the real-world situation we were modeling, using whichever of entity sets or relationships most closely matched the concept being modeled. There is also some advantage to replacing two concepts by one: the notation in which we express schemas is thereby simplified, and implementation techniques that make querying of the database more efficient can be applied to all sorts of data. We shall begin to appreciate these advantages of the relational model when we study implementation of the DBMS, starting in Chapter 11.

Now, let us consider the object-oriented model we introduced in Section 4.2. There are two principal concepts: the class (or its extent) and the relationship. Likewise, the object-relational model of Section 4.5 has two similar concepts: the attribute type (which includes classes) and the relation.

We may see the semistructured-data model as blending the two concepts, class-and-relationship or class-and-relation, much as the relational model blends entity sets and relationships. However, the motivation for the blending appears to be different in each case. While, as we mentioned, the relational model owes some of its success to the fact that it facilitates efficient implementation, interest in the semistructured-data model appears motivated primarily by its flexibility. While the other models seen so far each start from a notion of a schema — E/R diagrams, relation schemas, or ODL declarations, for instance — semistructured data is "schemaless." More properly, the data itself carries information about

The grant diagram is a useful way to remember enough about the history of grants and revocations to keep track of who has what privilege and from whom they obtained those privileges.

8.9 References for Chapter 8

Again, the reader is referred to the bibliographic notes of Chapter 6 for information on obtaining the SQL standards. The PSM standard is [4], and [5] is a comprehensive book on the subject. [6] is a popular reference on JDBC.

There is a discussion of problems with this standard in the area of transactions and cursors in [1]. More about transactions and how they are implemented can be found in the bibliographic notes to Chapter 18.

The ideas behind the SQL authorization mechanism originated in [3] and [2].

1. Berenson, H., P. A. Bernstein, J. N. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–10, 1995.

2. Fagin, R., "On an authorization mechanism," *ACM Transactions on Database Systems* **3**:3, pp. 310–319, 1978.

3. Griffiths, P. P. and B. W. Wade, "An authorization mechanism for a relational database system," *ACM Transactions on Database Systems* **1**:3, pp. 242–255, 1976.

4. ISO/IEC Report 9075-4, 1996.

5. Melton, J., *Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM*, Morgan-Kaufmann, San Francisco, 1998.

6. White, S., M. Fisher, R. Cattell, G. Hamilton, and M. Hapner, *JDBC API Tutorial and Reference*, Addison-Wesley, Boston, 1999.

Chapter 9

Object-Orientation in Query Languages

In this chapter, we shall discuss two ways in which object-oriented programming enters the world of query languages. OQL, or *Object Query Language*, is a standardized query language for object-oriented databases. It combines the high-level, declarative programming of SQL with the object-oriented programming paradigm. OQL is designed to operate on data described in ODL, the object-oriented data-description language that we introduced in Section 4.2.

If OQL is an attempt to bring the best of SQL into the object-oriented world, then the relatively new, object-relational features of the SQL-99 standard can be characterized as bringing the best of object-orientation into the relational world. In some senses, the two languages "meet in the middle," but there are differences in approach that make certain things easier in one language than the other.

In essence, the two approaches to object-orientation differ in their answer to the question: "how important is the relation?" For the object-oriented community centered around ODL and OQL, the answer is "not very." Thus, in OQL we find objects of all types, some of which are sets or bags of structures (i.e., relations). For the SQL community, the answer is that relations are still the fundamental data-structuring concept. In the object-relational approach that we introduced in Section 4.5, the relational model is extended by allowing more complex types for the tuples of relations and for attributes. Thus, objects and classes are introduced into the relational model, but always in the context of relations.

9.1 Introduction to OQL

OQL, the *Object Query Language*, gives us an SQL-like notation for expressing queries. It is intended that OQL will be used as an extension to some

object-oriented *host* language, such as C++, Smalltalk, or Java. Objects will be manipulated both by OQL queries and by the conventional statements of the host language. The ability to mix host-language statements and OQL queries without explicitly transferring values between the two languages is an advance over the way SQL is embedded into a host language, as was discussed in Section 8.1.

9.1.1 An Object-Oriented Movie Example

In order to illustrate the dictions of OQL, we need a running example. It will involve the familiar classes `Movie`, `Star`, and `Studio`. We shall use the definitions of `Movie`, `Star`, and `Studio` from Fig. 4.3, augmenting them with key and extent declarations. Only `Movie` has methods, gathered from Fig. 4.4. The complete example schema is in Fig. 9.1.

9.1.2 Path Expressions

We access components of objects and structures using a dot notation that is similar to the dot used in C and also related to the dot used in SQL. The general rule is as follows. If a denotes an object belonging to class C, and p is some property of the class — either an attribute, relationship, or method of the class — then $a.p$ denotes the result of "applying" p to a. That is:

1. If p is an attribute, then $a.p$ is the value of that attribute in object a.
2. If p is a relationship, then $a.p$ is the object or collection of objects related to a by relationship p.
3. If p is a method (perhaps with parameters), then $a.p(\cdots)$ is the result of applying p to a.

Example 9.1: Let `myMovie` denote an object of type `Movie`. Then:

- The value of `myMovie.length` is the length of the movie, that is, the value of the `length` attribute for the `Movie` object denoted by `myMovie`.
- The value of `myMovie.lengthInHours()` is a real number, the length of the movie in hours, computed by applying the method `lengthInHours` to object `myMovie`.
- The value of `myMovie.stars` is the set of `Star` objects related to the movie `myMovie` by the relationship `stars`.
- Expression `myMovie.starNames(myStars)` returns no value (i.e., in C++ the type of this expression is `void`). As a side effect, however, it sets the value of the output variable `myStars` of the method `starNames` to be a set of strings; those strings are the names of the stars of the movie.


```
class Movie
    (extent Movies key (title, year))
{
   attribute string title;
   attribute integer year;
   attribute integer length;
   attribute enum Film {color,blackAndWhite} filmType;
   relationship Set<Star> stars
                inverse Star::starredIn;
   relationship Studio ownedBy
                inverse Studio::owns;
   float lengthInHours() raises(noLengthFound);
   void starNames(out Set<String>);
   void otherMovies(in Star, out Set<Movie>)
                raises(noSuchStar);
};


class Star
    (extent Stars key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set<Movie> starredIn
                 inverse Movie::stars;
};


class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                 inverse Movie::ownedBy;
};
```

Figure 9.1: Part of an object-oriented movie database

Arrows and Dots

OQL allows the arrow -> as a synonym for the dot. This convention is partly in the spirit of C, where the dot and arrow both obtain components of a structure. However, in C, the arrow and dot operators have slightly different meanings; in OQL they are the same. In C, expression `a.f` expects `a` to be a structure, while `p->f` expects `p` to be a pointer to a structure. Both produce the value of the field `f` of that structure.

□

If it makes sense, we can form expressions with several dots. For example, if `myMovie` denotes a movie object, then `myMovie.ownedBy` denotes the `Studio` object that owns the movie, and `myMovie.ownedBy.name` denotes the string that is the name of that studio.

9.1.3 Select-From-Where Expressions in OQL

OQL permits us to write expressions using a select-from-where syntax similar to SQL's familiar query form. Here is an example asking for the year of the movie *Gone With the Wind*.

```
SELECT m.year
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

Notice that, except for the double-quotes around the string constant, this query could be SQL rather than OQL.

In general, the OQL select-from-where expression consists of:

1. The keyword `SELECT` followed by a list of expressions.
2. The keyword `FROM` followed by a list of one or more variable declarations. A variable is declared by giving
   (a) An expression whose value has a collection type, e.g. a set or bag.
   (b) The optional keyword `AS`, and
   (c) The name of the variable.

   Typically, the expression of (a) is the extent of some class, such as the extent `Movies` for class `Movie` in the example above. An extent is the analog of a relation in an SQL `FROM` clause. However, it is possible to use in a variable declaration any collection-producing expression, such as another select-from-where expression.

3. The keyword `WHERE` and a boolean-valued expression. This expression, like the expression following the `SELECT`, may only use as operands constants and those variables declared in the `FROM` clause. The comparison operators are like SQL's, except that `!=`, rather than `<>`, is used for "not equal to." The logical operators are `AND`, `OR`, and `NOT`, like SQL's.

The query produces a bag of objects. We compute this bag by considering all possible values of the variables in the `FROM` clause, in nested loops. If any combination of values for these variables satisfies the condition of the `WHERE` clause, then the object described by the `SELECT` clause is added to the bag that is the result of the select-from-where statement.

Example 9.2: Here is a more complex OQL query:

```
SELECT s.name
FROM Movies m, m.stars s
WHERE m.title = "Casablanca"
```

This query asks for the names of the stars of *Casablanca*. Notice the sequence of terms in the `FROM` clause. First we define `m` to be an arbitrary object in the class `Movie`, by saying `m` is in the extent of that class, which is `Movies`. Then, for each value of `m` we let `s` be a `Star` object in the set `m.stars` of stars of movie `m`. That is, we consider in two nested loops all pairs (`m`, `s`) such that `m` is a movie and `s` a star of that movie. The evaluation can be sketched as:

```
FOR each m in Movies DO
    FOR each s in m.stars DO
        IF m.title = "Casablanca" THEN
            add s.name to the output bag
```

The `WHERE` clause restricts our consideration to those pairs that have `m` equal to the `Movie` object whose title is *Casablanca*. Then, the `SELECT` clause produces the bag (which should be a set in this case) of all the name attributes of star objects `s` in the (`m`, `s`) pairs that satisfy the `WHERE` clause. These names are the names of the stars in the set m_c`.stars`, where m_c is the *Casablanca* movie object. □

9.1.4 Modifying the Type of the Result

A query like Example 9.2 produces a bag of strings as a result. That is, OQL follows the SQL default of not eliminating duplicates in its answer unless directed to do so. However, we can force the result to be a set or a list if we wish.

- To make the result a set, use the keyword `DISTINCT` after `SELECT`, as in SQL.

Alternative Form of FROM Lists

In addition to the SQL-style elements of FROM clauses, where the collection is followed by a name for a typical element, OQL allows a completely equivalent, more logical, yet less SQL-ish form. We can give the typical element name, then the keyword IN, and finally the name of the collection. For instance,

```
FROM m IN Movies, s IN m.stars
```

is an equivalent FROM clause for the query in Example 9.2.

- To make the result a list, add an ORDER BY clause at the end of the query, again as in SQL.

The following examples will illustrate the correct syntax.

Example 9.3: Let us ask for the names of the stars of Disney movies. The following query does the job, eliminating duplicate names in the situation where a star appeared in several Disney movies.

```
SELECT DISTINCT s.name
FROM Movies m, m.stars s
WHERE m.ownedBy.name = "Disney"
```

The strategy of this query is similar to that of Example 9.2. We again consider all pairs of a movie and a star of that movie in two nested loops as in Example 9.2. But now, the condition on that pair (m, s) is that "Disney" is the name of the studio whose Studio object is m.ownedBy. □

The ORDER BY clause in OQL is quite similar to the same clause in SQL. Keywords ORDER BY are followed by a list of expressions. The first of these expressions is evaluated for each object in the result of the query, and objects are ordered by this value. Ties, if any, are broken by the value of the second expression, then the third, and so on. By default, the order is ascending, but a choice of ascending or descending order can be indicated by the keyword ASC or DESC, respectively, following an attribute, as in SQL.

Example 9.4: Let us find the set of Disney movies, but let the result be a list of movies, ordered by length. If there are ties, let the movies of equal length be ordered alphabetically. The query is:

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
ORDER BY m.length, m.title
```


In the first three lines, we consider each Movie object m. If the name of the studio that owns this movie is "Disney," then the complete object m becomes a member of the output bag. The fourth line specifies that the objects m produced by the select-from-where query are to be ordered first by the value of m.length (i.e., the length of the movie) and then, if there are ties, by the value of m.title (i.e., the title of the movie). The value produced by this query is thus a list of Movie objects. □

9.1.5 Complex Output Types

The elements in the SELECT clause need not be simple variables. They can be any expression, including expressions built using type constructors. For example, we can apply the Struct type constructor to several expressions and get a select-from-where query that produces a set or bag of structures.

Example 9.5: Suppose we want the set of pairs of stars living at the same address. We can get this set with the query:

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
FROM Stars s1, Stars s2
WHERE s1.address = s2.address AND s1.name < s2.name
```

That is, we consider all pairs of stars, s1 and s2. The WHERE clause checks that they have the same address. It also checks that the name of the first star precedes the name of the second in alphabetic order, so we don't produce pairs consisting of the same star twice and we don't produce the same pair of stars in two different orders.

For every pair that passes the two tests, we produce a record structure. The type of this structure is a record with two fields, named star1 and star2. The type of each field is the class Star, since that is the type of the variables s1 and s2 that provide values for the two fields. That is, formally, the type of the structure is

```
Struct{star1: Star, star2: Star}
```

The type of the result of the query is a set of these structures, that is:

```
Set<Struct{star1: Star, star2: Star}>
```

□

9.1.6 Subqueries

We can use a select-from-where expression anywhere a collection is appropriate. We shall give one example: in the FROM clause. Several other examples of subquery use appear in Section 9.2.

SELECT Lists of Length One Are Special

Notice that when a SELECT list has only a single expression, the type of the result is a collection of values of the type of that expression. However, if we have more than one expression in the SELECT list, there is an implicit stucture formed with components for each expression. Thus, even had we started the query of Example 9.5 with

```
SELECT DISTINCT star1: s1, star2: s2
```

the type of the result would be

```
Set<Struct{star1: Star, star2: Star}>
```

However, in Example 9.3, the type of the result is `Set<String>`, not `Set<Struct{name: String}>`.

In the FROM clause, we may use a subquery to form a collection. We then allow a variable representing a typical element of that collection to range over each member of the collection.

Example 9.6: Let us redo the query of Example 9.3, which asked for the stars of the movies made by Disney. First, the set of Disney movies could be obtained by the query, as was used in Example 9.4.

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
```

We can now use this query as a subquery to define the set over which a variable d, representing the Disney movies, can range.

```
SELECT DISTINCT s.name
FROM (SELECT m
      FROM Movies m
      WHERE m.ownedBy.name = "Disney") d,
     d.stars s
```

This expression of the query "Find the stars of Disney movies" is no more succinct than that of Example 9.3, and perhaps less so. However, it does illustrate a new form of building queries available in OQL. In the query above, the FROM clause has two nested loops. In the first, the variable d ranges over all Disney movies, the result of the subquery in the FROM clause. In the second loop, nested within the first, the variable s ranges over all stars of the Disney movie d. Notice that no WHERE clause is needed in the outer query. □

9.1.7 Exercises for Section 9.1

Exercise 9.1.1: In Fig. 9.2 is an ODL description of our running products exercise. We have made each of the three types of products subclasses of the main `Product` class. The reader should observe that a type of a product can be obtained either from the attribute `type` or from the subclass to which it belongs. This arrangement is not an excellent design, since it allows for the possibility that, say, a PC object will have its `type` attribute equal to `"laptop"` or `"printer"`. However, the arrangement gives you some interesting options regarding how one expresses queries.

Because `type` is inherited by `Printer` from the superclass `Product`, we have had to rename the `type` attribute of `Printer` to be `printerType`. The latter attribute gives the process used by the printer (e.g., laser or inkjet), while `type` of `Product` will have values such as PC, laptop, or printer.

Add to the ODL code of Fig. 9.2 method signatures (see Section 4.2.7) appropriate for functions that do the following:

* a) Subtract x from the price of a product. Assume x is provided as an input parameter of the function.

* b) Return the speed of a product if the product is a PC or laptop and raise the exception `notComputer` if not.

c) Set the screen size of a laptop to a specified input value x.

! d) Given an input product p, determine whether the product q to which the method is applied has a higher speed and a lower price than p. Raise the exception `badInput` if p is not a product with a speed (i.e., neither a PC nor laptop) and the exception `noSpeed` if q is not a product with a speed.

Exercise 9.1.2: Using the ODL schema of Exercise 9.1.1 and Fig. 9.2, write the following queries in OQL:

* a) Find the model numbers of all products that are PC's with a price under $2000.

b) Find the model numbers of all the PC's with at least 128 megabytes of RAM.

*! c) Find the manufacturers that make at least two different models of laser printer.

d) Find the set of pairs (r, h) such that some PC or laptop has r megabytes of RAM and h gigabytes of hard disk.

e) Create a list of the PC's (objects, not model numbers) in ascending order of processor speed.

! f) Create a list of the model numbers of the laptops with at least 64 megabytes of RAM, in descending order of screen size.

```
class Product
    (extent Products
     key model)
 {
    attribute integer model;
    attribute string manufacturer;
    attribute string type;
    attribute real price;
 };


class PC extends Product
    (extent PCs)
 {
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
    attribute string rd;
 };


class Laptop extends Product
    (extent Laptops)
 {
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
    attribute real screen;
 };


class Printer extends Product
    (extent Printers)
 {
    attribute boolean color;
    attribute string printerType;
 };
```

Figure 9.2: Product schema in ODL


```
class Class
    (extent Classes
     key name)
 {
    attribute string name;
    attribute string country;
    attribute integer numGuns;
    attribute integer bore;
    attribute integer displacement;
    relationship Set<Ship> ships inverse Ship::classOf;
 };

class Ship
    (extent Ships
     key name)
 {
    attribute string name;
    attribute integer launched;
    relationship Class classOf inverse Class::ships;
    relationship Set<Outcome> inBattles
                 inverse Outcome::theShip;
 };

class Battle
    (extent Battles
     key name)
 {
    attribute string name;
    attribute Date dateFought;
    relationship Set<Outcome> results
                 inverse Outcome::theBattle;
 };

class Outcome
    (extent Outcomes)
 {
    attribute enum Stat {ok,sunk,damaged} status;
    relationship Ship theShip inverse Ship::inBattles;
    relationship Battle theBattle inverse Battle::results;
 };
```

Figure 9.3: Battleships database in ODL

Exercise 9.1.3: In Fig. 9.3 is an ODL description of our running "battleships" database. Add the following method signatures:

a) Compute the firepower of a ship, that is, the number of guns times the cube of the bore.

b) Find the sister ships of a ship. Raise the exception `noSisters` if the ship is the only one of its class.

c) Given a battle b as a parameter, and applying the method to a ship s, find the ships sunk in the battle b, provided s participated in that battle. Raise the exception `didNotParticipate` if ship s did not fight in battle b.

d) Given a name and a year launched as parameters, add a ship of this name and year to the class to which the method is applied.

! **Exercise 9.1.4:** Repeat each part of Exercise 9.1.2 using at least one subquery in each of your queries.

Exercise 9.1.5: Using the ODL schema of Exercise 9.1.3 and Fig. 9.3, write the following queries in OQL:

a) Find the names of the classes of ships with at least nine guns.

b) Find the ships (objects, not ship names) with at least nine guns.

c) Find the names of the ships with a displacement under 30,000 tons. Make the result a list, ordered by earliest launch year first, and if there are ties, alphabetically by ship name.

d) Find the pairs of objects that are sister ships (i.e., ships of the same class). Note that the objects themselves are wanted, not the names of the ships.

! e) Find the names of the battles in which ships of at least two different countries were sunk.

!! f) Find the names of the battles in which no ship was listed as damaged.

9.2 Additional Forms of OQL Expressions

In this section we shall see some of the other operators, besides select-from-where, that OQL provides to help us build expressions. These operators include logical quantifiers — for-all and there-exists — aggregation operators, the group-by operator, and set operators — union, intersection, and difference.

9.2.1 Quantifier Expressions

We can test whether all members of a collection satisfy some condition, and we can test whether at least one member of a collection satisfies a condition. To test whether all members x of a collection S satisfy condition $C(x)$, we use the OQL expression:

`FOR ALL` x `IN` S : $C(x)$

The result of this expression is `TRUE` if every x in S satisfies $C(x)$ and is `FALSE` otherwise. Similarly, the expression

`EXISTS` x `IN` S : $C(x)$

has value `TRUE` if there is at least one x in S such that $C(x)$ is `TRUE` and it has value `FALSE` otherwise.

Example 9.7: Another way to express the query "find all the stars of Disney movies" is shown in Fig. 9.4. Here, we focus on a star s and ask if they are the star of some movie m that is a Disney movie. Line (3) tells us to consider all movies m in the set of movies `s.starredIn`, which is the set of movies in which star s appeared. Line (4) then asks whether movie m is a Disney movie. If we find even one such movie m, the value of the `EXISTS` expression in lines (3) and (4) is `TRUE`; otherwise it is `FALSE`. □

```
1)  SELECT s
2)  FROM Stars s
3)  WHERE EXISTS m IN s.starredIn :
4)     m.ownedBy.name = "Disney"
```

Figure 9.4: Using an existential subquery

Example 9.8: Let us use the for-all operator to write a query asking for the stars that have appeared only in Disney movies. Technically, that set includes "stars" who appear in no movies at all (as far as we can tell from our database). It is possible to add another condition to our query, requiring that the star appear in at least one movie, but we leave that improvement as an exercise. Figure 9.5 shows the query. □

9.2.2 Aggregation Expressions

OQL uses the same five aggregation operators that SQL does: `AVG`, `COUNT`, `SUM`, `MIN`, and `MAX`. However, while these operators in SQL may be thought of as

```
SELECT s
FROM Stars s
WHERE FOR ALL m IN s.starredIn :
    m.ownedBy.name = "Disney"
```

Figure 9.5: Using a subquery with universal quantification

applying to a designated column of a table, the same operators in OQL apply to all collections whose members are of a suitable type. That is, COUNT can apply to any collection; SUM and AVG can be applied to collections of arithmetic types such as integers, and MIN and MAX can be applied to collections of any type that can be compared, e.g., arithmetic values or strings.

Example 9.9: To compute the average length of all movies, we need to create a bag of all movie lengths. Note that we don't want the *set* of movie lengths, because then two movies that had the same length would count as one. The query is:

```
AVG(SELECT m.length FROM Movies m)
```

That is, we use a subquery to extract the length components from movies. Its result is the bag of lengths of movies, and we apply the AVG operator to this bag, giving the desired answer. □

9.2.3 Group-By Expressions

The GROUP BY clause of SQL carries over to OQL, but with an interesting twist in perspective. The form of a GROUP BY clause in OQL is:

1. The keywords GROUP BY.
2. A comma-separated list of one or more *partition attributes*. Each of these consists of
   (a) A field name,
   (b) A colon, and
   (c) An expression.

That is, the form of a GROUP BY clause is:

$$\texttt{GROUP BY } f_1\texttt{:}e_1,\ f_2\texttt{:}e_2, \ldots, f_n\texttt{:}e_n$$

Each GROUP BY clause follows a select-from-where query. The expressions $e_1, e_2, \ldots, e_n$ may refer to variables mentioned in the FROM clause. To facilitate the explanation of how GROUP BY works, let us restrict ourselves to the common

case where there is only one variable x in the FROM clause. The value of x ranges over some collection, C. For each member of C, say i, that satisfies the condition of the WHERE clause, we evaluate all the expressions that follow the GROUP BY, to obtain values $e_1(i), e_2(i), \ldots, e_n(i)$. This list of values is the group to which value i belongs.

The Intermediate Collection

The actual value returned by the GROUP BY is a set of structures, which we shall call the *intermediate collection*. The members of the intermediate collection have the form

$$\texttt{Struct(}f_1\texttt{:}v_1,\ f_2\texttt{:}v_2, \ldots, f_n\texttt{:}v_n,\ \texttt{partition:}P\texttt{)}$$

The first n fields indicate the group. That is, $(v_1, v_2, \ldots, v_n)$ must be the list of values $(e_1(i), e_2(i), \ldots, e_n(i))$ for at least one value of i in the collection C that meets the condition of the WHERE clause.

The last field has the special name partition. Its value P is, intuitively, the values i that belong in this group. More precisely, P is a bag consisting of structures of the form Struct(x:i), where x is the variable of the FROM clause.

The Output Collection

The SELECT clause of a select-from-where expression that has a GROUP BY clause may refer only to the fields in the structures of the intermediate collection, namely $f_1, f_2, \ldots, f_n$ and partition. Through partition, we may refer to the field x that is present in the structures that are members of the bag P that forms the value of partition. Thus, we may refer to the variable x that appears in the FROM clause, but we may only do so within an aggregation operator that aggregates over all the members of a bag P. The result of the SELECT clause will be referred to as the *output collection*.

Example 9.10: Let us build a table of the total length of movies for each studio and for each year. In OQL, what we actually construct is a bag of structures, each with three components — a studio, a year, and the total length of movies for that studio and year. The query is shown in Fig. 9.6.

```
SELECT stdo, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY stdo: m.ownedBy.name, yr: m.year
```

Figure 9.6: Grouping movies by studio and year

To understand this query, let us start at the FROM clause. There, we find that variable m ranges over all Movie objects. Thus, m here plays the role of x

in our general discussion. In the GROUP BY clause are two fields stdo and yr, corresponding to the expressions m.ownedBy.name and m.year, respectively.

For instance, *Pretty Woman* is a movie made by Disney in 1990. When m is the object for this movie, the value of m.ownedBy.name is "Disney" and the value of m.year is 1990. As a result, the intermediate collection has, as one member, the structure:

$$\texttt{Struct(stdo:"Disney", yr:1990, partition:}P\texttt{)}$$

Here, P is a set of structures. It contains, for example,

$$\texttt{Struct(m:}m_{pw}\texttt{)}$$

where m_{pw} is the Movie object for *Pretty Woman*. Also in P are one-component structures with field name m for every other Disney movie of 1990.

Now, let us examine the SELECT clause. For each structure in the intermediate collection, we build one structure that is in the output collection. The first component of each output structure is stdo. That is, the field name is stdo and its value is the value of the stdo field of the corresponding structure in the intermediate collection. Similarly, the second component of the result has field name yr and a value equal to the yr component of the intermediate collection.

The third component of each structure in the output is

```
SUM(SELECT p.m.length FROM partition p)
```

To understand this select-from expression we first realize that variable p ranges over the members of the partition field of the structure in the GROUP BY result. Each value of p, recall, is a structure of the form Struct(m:o), where o is a movie object. The expression p.m therefore refers to this object o. Thus, p.m.length refers to the length component of this Movie object.

As a result, the select-from query produces the bag of lengths of the movies in a particular group. For instance, if stdo has the value "Disney" and yr has the value 1990, then the result of the select-from is the bag of the lengths of the movies made by Disney in 1990. When we apply the SUM operator to this bag we get the sum of the lengths of the movies in the group. Thus, one member of the output collection might be

```
Struct(stdo:"Disney", yr:1990, sumLength:1234)
```

if 1234 is the correct total length of all the Disney movies of 1990. □

Grouping When the FROM Clause has Multiple Collections

In the event that there is more than one variable in the FROM clause, a few changes to the interpretation of the query are necessary, but the principles remain the same as in the one-variable case above. Suppose that the variables appearing in the FROM clause are $x_1, x_2, \ldots, x_k$. Then:

1. All variables $x_1, x_2, \ldots, x_k$ may be used in the expressions $e_1, e_2, \ldots, e_n$ of the GROUP BY clause.

2. Structures in the bag that is the value of the partition field have fields named $x_1, x_2, \ldots, x_k$.

3. Suppose $i_1, i_2, \ldots, i_k$ are values for variables $x_1, x_2, \ldots, x_k$, respectively, that make the WHERE clause true. Then there is a structure in the intermediate collection of the form

   $$\texttt{Struct(}f_1\texttt{:}e_1(i_1,\ldots,i_k),\ldots,f_n\texttt{:}e_n(i_1,\ldots,i_k)\texttt{, partition:P)}$$

   and in bag P is the structure:

   $$\texttt{Struct(}x_1\texttt{:}i_1\texttt{, }x_2\texttt{:}i_2,\ldots,x_k\texttt{:}i_k\texttt{)}$$

9.2.4 HAVING Clauses

A GROUP BY clause of OQL may be followed by a HAVING clause, with a meaning like that of SQL's HAVING clause. That is, a clause of the form

```
HAVING <condition>
```

serves to eliminate some of the groups created by the GROUP BY. The condition applies to the value of the partition field of each structure in the intermediate collection. If true, then this structure is processed as in Section 9.2.3, to form a structure of the output collection. If false, then this structure does not contribute to the output collection.

Example 9.11: Let us repeat Example 9.10, but ask for the sum of the lengths of movies for only those studios and years such that the studio produced at least one movie of over 120 minutes. The query of Fig. 9.7 does the job. Notice that in the HAVING clause we used the same query as in the SELECT clause to obtain the bag of lengths of movies for a given studio and year. In the HAVING clause, we take the maximum of those lengths and compare it to 120. □

```
SELECT stdo, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY stdo: m.ownedBy.name, yr: m.year
HAVING MAX(SELECT p.m.length FROM partition p) > 120
```

Figure 9.7: Restricting the groups considered

9.2.5 Union, Intersection, and Difference

We may apply the union, intersection, and difference operators to two objects of set or bag type. These three operators are represented, as in SQL, by the keywords `UNION`, `INTERSECT`, and `EXCEPT`, respectively.

```
1)      (SELECT DISTINCT m
2)       FROM Movies m, m.stars s
3)       WHERE s.name = "Harrison Ford")
4)  EXCEPT
5)      (SELECT DISTINCT m
6)       FROM Movies m
7)       WHERE m.ownedBy.name = "Disney")
```

Figure 9.8: Query using the difference of two sets

Example 9.12: We can find the set of movies starring Harrison Ford that were not made by Disney with the difference of two select-from-where queries shown in Fig. 9.8. Lines (1) through (3) find the set of movies starring Ford, and lines (5) through (7) find the set of movies made by Disney. The `EXCEPT` at line (4) takes their difference. □

We should notice the `DISTINCT` keywords in lines (1) and (5) of Fig. 9.8. This keyword forces the results of the two queries to be of set type; without `DISTINCT`, the result would be of bag (*multiset*) type. In OQL, the operators `UNION`, `INTERSECT`, and `EXCEPT` operate on either sets or bags. When both arguments are sets, then the operators have their usual set meaning.

However, when both arguments are of bag type, or one is a bag and one is a set, then the bag meaning of the operators is used. Recall Section 5.3.2, where the definitions of union, intersection, and difference for bags was explained.

For the particular query of Fig. 9.8, the number of times a movie appears in the result of either subquery is zero or one, so the result is the same regardless of whether `DISTINCT` is used. However, the *type* of the result differs. If `DISTINCT` is used, then the type of the result is `Set<Movie>`, while if `DISTINCT` is omitted in one or both places, then the result is of type `Bag<Movie>`.

9.2.6 Exercises for Section 9.2

Exercise 9.2.1: Using the ODL schema of Exercise 9.1.1 and Fig. 9.2, write the following queries in OQL:

* a) Find the manufacturers that make both PC's and printers.

* b) Find the manufacturers of PC's, all of whose PC's have at least 20 gigabytes of hard disk.

c) Find the manufacturers that make PC's but not laptops.

* d) Find the average speed of PC's.

* e) For each CD or DVD speed, find the average amount of RAM on a PC.

! f) Find the manufacturers that make some product with at least 64 megabytes of RAM and also make a product costing under $1000.

!! g) For each manufacturer that makes PC's with an average speed of at least 1200, find the maximum amount of RAM that they offer on a PC.

Exercise 9.2.2: Using the ODL schema of Exercise 9.1.3 and Fig. 9.3, write the following queries in OQL:

a) Find those classes of ship all of whose ships were launched prior to 1919.

b) Find the maximum displacement of any class.

! c) For each gun bore, find the earliest year in which any ship with that bore was launched.

*!! d) For each class of ships at least one of which was launched prior to 1919, find the number of ships of that class sunk in battle.

! e) Find the average number of ships in a class.

! f) Find the average displacement of a ship.

!! g) Find the battles (objects, not names) in which at least one ship from Great Britain took part and in which at least two ships were sunk.

! **Exercise 9.2.3:** We mentioned in Example 9.8 that the OQL query of Fig. 9.5 would return stars who starred in no movies at all, and therefore, technically appeared "only in Disney movies." Rewrite the query to return only those stars who have appeared in at least one movie and all movies in which they appeared where Disney movies.

! **Exercise 9.2.4:** Is it ever possible for `FOR ALL` x `IN` S : $C(x)$ to be true, while `EXISTS` x `IN` S : $C(x)$ is false? Explain your reasoning.

9.3 Object Assignment and Creation in OQL

In this section we shall consider how OQL connects to its host language, which we shall take to be C++ in examples, although another object-oriented, general-purpose programming language (e.g. Java) might be the host language in some systems.

9.3.1 Assigning Values to Host-Language Variables

Unlike SQL, which needs to move data between components of tuples and host-language variables, OQL fits naturally into its host language. That is, the expressions of OQL that we have learned, such as select-from-where, produce objects as values. It is possible to assign to any host-language variable of the proper type a value that is the result of one of these OQL expressions.

Example 9.13: The OQL expression

```
SELECT DISTINCT m
FROM Movies m
WHERE m.year < 1920
```

produces the set of all those movies made before 1920. Its type is `Set<Movie>`. If `oldMovies` is a host-language variable of the same type, then we may write (in C++ extended with OQL):

```
oldMovies = SELECT DISTINCT m
            FROM Movies m
            WHERE m.year < 1920;
```

and the value of `oldMovies` will become the set of these `Movie` objects. □

9.3.2 Extracting Elements of Collections

Since the select-from-where and group-by expressions each produce collections — either sets, bags, or lists — we must do something extra if we want a single element of that collection. This statement is true even if we have a collection that we are sure contains only one element. OQL provides the operator `ELEMENT` to turn a singleton collection into its lone member. This operator can be applied, for instance, to the result of a query that is known to return a singleton.

Example 9.14: Suppose we would like to assign to the variable `gwtw`, of type `Movie` (i.e., the `Movie` class is its type) the object representing the movie *Gone With the Wind*. The result of the query

```
SELECT m
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

is the bag containing just this one object. We cannot assign this bag to variable `gwtw` directly, because we would get a type error. However, if we apply the `ELEMENT` operator first,

```
gwtw = ELEMENT(SELECT m
               FROM Movies m
               WHERE m.title = "Gone With the Wind"
       );
```


then the type of the variable and the expression match, and the assignment is legal. □

9.3.3 Obtaining Each Member of a Collection

Obtaining each member of a set or bag is more complex, but still simpler than the cursor-based algorithms we needed in SQL. First, we need to turn our set or bag into a list. We do so with a select-from-where expression that uses `ORDER BY`. Recall from Section 9.1.4 that the result of such an expression is a list of the selected objects or values.

Example 9.15: Suppose we want a list of all the movie objects in the class `Movie`. We can use the title and (to break ties) the year of the movie, since `(title, year)` is a key for `Movie`. The statement

```
movieList = SELECT m
            FROM Movies m
            ORDER BY m.title, m.year;
```

assigns to host-language variable `movieList` a list of all the `Movie` objects, sorted by title and year. □

Once we have a list, sorted or not, we can access each element by number; the ith element of the list L is obtained by $L[i-1]$. Note that lists and arrays are assumed numbered starting at 0, as in C or C++.

Example 9.16: Suppose we want to write a C++ function that prints the title, year, and length of each movie. A sketch of the function is shown in Fig. 9.9.

```
1)  movieList = SELECT m
                FROM Movies m
                ORDER BY m.title, m.year;
2)  numberOfMovies = COUNT(Movies);
3)  for(i=0; i<numberOfMovies; i++) {
4)      movie = movieList[i];
5)      cout << movie.title << " " << movie.year << " "
6)           << movie.length << "\n";
    }
```

Figure 9.9: Examining and printing each movie

Line (1) sorts the `Movie` class, placing the result into variable `movieList`, whose type is `List<Movie>`. Line (2) computes the number of movies, using the OQL operator `COUNT`. Lines (3) through (6) are a for-loop in which integer

variable `i` ranges over each position of the list. For convenience, the ith element of the list is assigned to variable `movie`. Then, at lines (5) and (6) the relevant attributes of the movie are printed. □

9.3.4 Constants in OQL

Constants in OQL (sometimes referred to as *immutable objects*) are constructed from a basis and recursive constructors, in a manner analogous to the way ODL types are constructed.

1. *Basic values*, which are either
   (a) *Atomic values*: integers, floats, characters, strings, and booleans. These are represented as in SQL, with the exception that double-quotes are used to surround strings.
   (b) *Enumerations*. The values in an enumeration are actually declared in ODL. Any one of these values may be used as a constant.
2. *Complex values* built using the following type constructors:
   (a) `Set(...)`.
   (b) `Bag(...)`.
   (c) `List(...)`.
   (d) `Array(...)`.
   (e) `Struct(...)`.

   The first four of these are called *collection types*. The collection types and `Struct` may be applied at will to any values of the appropriate type(s), basic or complex. However, when applying the `Struct` operator, one needs to specify the field names and their corresponding values. Each field name is followed by a colon and the value, and field-value pairs are separated by commas. Note that the same type constructors are used in ODL, but here we use round, rather than triangular, brackets.

Example 9.17: The expression `Bag(2,1,2)` denotes the bag in which integer 2 appears twice and integer 1 appears once. The expression

```
Struct(foo: bag(2,1,2), bar: "baz")
```

denotes a structure with two fields. Field `foo`, has the bag described above as its value, and `bar`, has the string `"baz"` for its value. □

9.3.5 Creating New Objects

We have seen that OQL expressions such as select-from-where allow us to create new objects. It is also possible to create objects by assembling constants or other expressions into structures and collections explicitly. We saw an example of this convention in Example 9.5, where the line

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
```

was used to specify that the result of the query is a set of objects whose type is `Struct{star1: Star, star2: Star}`. We gave the field names `star1` and `star2` to specify the structure, while the types of these fields could be deduced from the types of the variables `s1` and `s2`.

Example 9.18: The construction of constants that we saw in Section 9.3.4 can be used with assignments to variables, in a manner similar to that of other programming languages. For instance, consider the following sequence of assignments:

```
x = Struct(a:1, b:2);
y = Bag(x, x, Struct(a:3, b:4));
```

The first line gives variable `x` a value of type

```
Struct(a:integer, b:integer)
```

a structure with two integer-valued fields named `a` and `b`. We may represent values of this type as pairs, with just the integers as components and not the field names `a` and `b`. Thus, the value of x may be represented by $(1,2)$. The second line defines `y` to be a bag whose members are structures of the same type as x, above. The pair $(1,2)$ appears twice in this bag, and $(3,4)$ appears once. □

Classes or other defined types can have instances created by *constructor functions*. Classes typically have several different forms of constructor functions, depending on which properties are initialized explicitly and which are given some default value. For example, methods are not initialized, most attributes will get initial values, and relationships might be initialized to the empty set and augmented later. The name for each of these constructor functions is the name of the class, and they are distinguished by the field names mentioned in their arguments. The details of how these constructor functions are defined depend on the host language.

Example 9.19: Let us consider a possible constructor function for `Movie` objects. This function, we suppose, takes values for the attributes `title`, `year`, `length`, and `ownedBy`, producing an object that has these values in the listed fields and an empty set of stars. Then, if `mgm` is a variable whose value is the MGM `Studio` object, we might create a *Gone With the Wind* object by:

```
gwtw = Movie(title: "Gone With the Wind",
             year: 1939,
             length: 239,
             ownedBy: mgm);
```

This statement has two effects:

1. It creates a new `Movie` object, which becomes part of the extent `Movies`.
2. It makes this object the value of host-language variable `gwtw`.

□

9.3.6 Exercises for Section 9.3

Exercise 9.3.1: Assign to a host-language variable x the following constants:

* a) The set $\{1, 2, 3\}$.

b) The bag $\{1, 2, 3, 1\}$.

c) The list $(1, 2, 3, 1)$.

d) The structure whose first component, named a, is the set $\{1, 2\}$ and whose second component, named b, is the bag $\{1, 1\}$.

e) The bag of structures, each with two fields named a and b. The respective pairs of values for the three structures in the bag are $(1, 2)$, $(2, 1)$, and $(1, 2)$.

Exercise 9.3.2: Using the ODL schema of Exercise 9.1.1 and Fig. 9.2, write statements of C++ (or an object-oriented host language of your choice) extended with OQL to do the following:

* a) Assign to host-language variable x the object for the PC with model number 1000.

b) Assign to host-language variable y the set of all laptop objects with at least 64 megabytes of RAM.

c) Assign to host-language variable z the average speed of PC's selling for less than \$1500.

! d) Find all the laser printers, print a list of their model numbers and prices, and follow it by a message indicating the model number with the lowest price.

!! e) Print a table giving, for each manufacturer of PC's, the minimum and maximum price.

Exercise 9.3.3: In this exercise, we shall use the ODL schema of Exercise 9.1.3 and Fig. 9.3. We shall assume that for each of the four classes of that schema, there is a constructor function of the same name that takes values for each of the attributes and single-valued relationships, but not the multivalued relationships, which are initialized to be empty. For the single-valued relationships to other classes, you may postulate a host-language variable whose current value is the related object. Create the following objects and assign the object to be the value of a host-language variable in each case.

* a) The battleship Colorado of the Maryland class, launched in 1923.

b) The battleship Graf Spee of the Lützow class, launched in 1936.

c) An outcome of the battle of Malaya was that the battleship Prince of Wales was sunk.

d) The battle of Malaya was fought Dec. 10, 1941.

e) The Hood class of British battlecruisers had eight 15-inch guns and a displacement of 41,000 tons.

9.4 User-Defined Types in SQL

We now turn to the way SQL-99 incorporates many of the object-oriented features that we have seen in ODL and OQL. Because of these recent extensions to SQL, a DBMS that follows this standard is often referred to as "object-relational." We met many of the object-relational concepts abstractly in Section 4.5. Now, it is time for us to study the details of the standard.

OQL has no specific notion of a relation: it is just a set (or bag) of structures. However, the relation is so central to SQL that objects in SQL keep relations as the core concept. The classes of ODL are transmogrified into *user-defined types*, or UDT's, in SQL. We find UDT's used in two distinct ways:

1. A UDT can be the type of a table.
2. A UDT can be the type of an attribute belonging to some table.

9.4.1 Defining Types in SQL

A user-defined type declaration in SQL can be thought of as roughly analogous to a class declaration in ODL, with some distinctions. First, key declarations for a relation with a user-defined type are part of the table definition, not the type definition; that is, many SQL relations can be declared to have the same (user-defined) type but different keys and other constraints. Second, in SQL we do not treat relationships as properties. A relationship must be represented by a separate relation, as was discussed in Section 4.4.5. A simple form of UDT definition is:

1. The keywords CREATE TYPE,
2. A name for the type,
3. The keyword AS,
4. A parenthesized, comma-separated list of attributes and their types.
5. A comma-separated list of methods, including their argument type(s), and return type.

That is, the definition of a type T has the form

CREATE TYPE T AS <attribute and method declarations> ;

Example 9.20: We can create a type representing movie stars, analogous to the class Star found in the OQL example of Fig. 9.1. However, we cannot represent directly a set of movies as a field within Star tuples. Thus, we shall start with only the name and address components of Star tuples.

To begin, note that the type of an address in Fig. 9.1 is itself a tuple, with components street and city. Thus, we need two type definitions, one for addresses and the other for stars. The necessary definitions are shown in Fig. 9.10.

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
);

CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

Figure 9.10: Two type definitions

A tuple of type AddressType has two components, whose attributes are street and city. The types of these components are character strings of length 50 and 20, respectively. A tuple of type StarType also has two components. The first is attribute name, whose type is a 30-character string, and the second is address, whose type is itself a UDT AddressType. that is, a tuple with street and city components. □

9.4.2 Methods in User-Defined Types

The declaration of a method resembles the way a function in PSM is introduced; see Section 8.2.1. There is no analog of PSM procedures as methods. That is, every method returns a value of some type. While function declarations and definitions in PSM are combined, a method needs both a declaration, within the definition of its type, and a separate definition, in a CREATE METHOD statement.

A method declaration looks like a PSM function declaration, with the keyword METHOD replacing CREATE FUNCTION. However, SQL methods typically have no arguments; they are applied to rows, just as ODL methods are applied to objects. In the definition of the method, SELF refers to this tuple, if necessary.

Example 9.21: Let us extend the definition of the type AddressType of Fig. 9.10 with a method houseNumber that extracts from the street component the portion devoted to the house address. For instance, if the street component were '123 Maple St.', then houseNumber should return '123'. The revised type definition is thus:

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
    )
    METHOD houseNumber() RETURNS CHAR(10);
```

We see the keyword METHOD, followed by the name of the method and a parenthesized list of its arguments and their types. In this case, there are no arguments, but the parentheses are still needed. Had there been arguments, they would have appeared, followed by their types, such as (a INT, b CHAR(5)). □

Separately, we need to define the method. A simple form of method definition consists of:

1. The keywords CREATE METHOD.
2. The method name, arguments and their types, and the RETURNS clause, as in the declaration of the method.
3. The keyword FOR and the name of the UDT in which the method is declared.
4. The body of the method. which is written in the same language as the bodies of PSM functions.

For instance, we could define the method houseNumber from Example 9.21 as:

```
CREATE METHOD houseNumber() RETURNS CHAR(10)
FOR AddressType
```

```
BEGIN
    ...
END;
```

We have omitted the body of the method because accomplishing the intended separation of the string `string` as intended is nontrivial, even in PSM.

9.4.3 Declaring Relations with a UDT

Having declared a type, we may declare one or more relations whose tuples are of that type. The form of relation declarations is like that of Section 6.6.2, but we use

OF <type name>

in place of the list of attribute declarations in a normal SQL table declaration. Other elements of a table declaration, such as keys, foreign keys, and tuple-based constraints, may be added to the table declaration if desired, and apply only to this table, not to the UDT itself.

Example 9.22: We could declare `MovieStar` to be a relation whose tuples were of type `StarType` by

```
CREATE TABLE MovieStar OF StarType;
```

As a result, table `MovieStar` has two attributes, `name` and `address`. The first attribute, `name`, is an ordinary character string, but the second, `address`, has a type that is itself a UDT, namely the type `AddressType`. □

It is common to have one relation for each type, and to think of that relation as the extent (in the sense of Section 4.3.4) of the class corresponding to that type. However, it is permissible to have many relations or none of a given type.

9.4.4 References

The effect of object identity in object-oriented languages is obtained in SQL through the notion of a *reference*. Tables whose type is a UDT may have a *reference column* that serves as its "identity." This column could be the primary key of the table, if there is one, or it could be a column whose values are generated and maintained unique by the DBMS, for example. We shall defer the matter of defining reference columns until we first see how reference types are used.

To refer to the tuples of a table with a reference column, an attribute may have as its type a reference to another type. If T is a UDT, then `REF(`T`)` is the type of a reference to a tuple of type T. Further, the reference may be given a *scope*, which is the name of the relation whose tuples are referred to. Thus, an attribute A whose values are references to tuples in relation R, where R is a table whose type is the UDT T, would be declared by:

A `REF(`T`) SCOPE` R

If no scope is specified, the reference can go to any relation of type T.

Example 9.23: Reference attributes are not sufficient to record in `MovieStar` the set of all movies they starred in, but they let us record the best movie for each star. Assume that we have declared a relation `Movie`, and that the type of this relation is the UDT `MovieType`; we shall define both `MovieType` and `Movie` later, in Fig. 9.11. The following is a new definition of `StarType` that includes an attribute `bestMovie` that is a reference to a movie.

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

Now, if relation `MovieStar` is defined to have the UDT above, then each star tuple will have a component that refers to a `Movie` tuple — the star's best movie. □

Next, we must arrange that a table such as `Movie` in Example 9.23 will have a reference column. Such a table is said to be *referenceable*. In a `CREATE TABLE` statement where the type of the table is a UDT (as in Section 9.4.3), we may append a clause of the form:

REF IS <attribute name> <how generated>

The attribute name is a name given to the column that will serve as an "object identifier" for tuples. The "how generated" clause is typically either:

1. `SYSTEM GENERATED`, meaning that the DBMS is responsible for maintaining a unique value in this column of each tuple, or
2. `DERIVED`, meaning that the DBMS will use the primary key of the relation to produce unique values for this column.

Example 9.24: Figure 9.11 shows how the UDT `MovieType` and relation `Movie` could be declared so that `Movie` is referenceable. The UDT is declared in lines (1) through (4). Then the relation `Movie` is defined to have this type in lines (5) through (7). Notice that we have declared `title` and `year`, together, to be the key for relation `Movie` in line (7).

We see in line (6) that the name of the "identity" column for `Movie` is `movieID`. This attribute, which automatically becomes a fourth attribute of `Movie`, along with `title`, `year`, and `inColor`, may be used in queries like any other attribute of `Movie`.

Line (6) also says that the DBMS is responsible for generating the value of `movieID` each time a new tuple is inserted into `Movie`. Had we replaced "`SYSTEM`

```
1)  CREATE TYPE MovieType AS (
2)      title   CHAR(30),
3)      year    INTEGER,
4)      inColor BOOLEAN
    );

5)  CREATE TABLE Movie OF MovieType (
6)      REF IS movieID SYSTEM GENERATED,
7)      PRIMARY KEY (title, year)
    );
```

Figure 9.11: Creating a referenceable table

GENERATED" by "DERIVED," then new tuples would get their value of movieID by some calculation, performed by the system, on the values of the primary-key attributes title and year from the same tuple. □

Example 9.25: Now, let us see how to represent the many-many relationship between movies and stars using references. Previously, we represented this relationship by a relation like StarsIn that contains tuples with the keys of Movie and MovieStar. As an alternative, we may define StarsIn to have references to tuples from these two relations.

First, we need to redefine MovieStar so it is a referenceable table, thusly:

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

Then, we may declare the relation StarsIn to have two attributes, which are references, one to a movie tuple and one to a star tuple. Here is a direct definition of this relation:

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

Optionally, we could have defined a UDT as above, and then declared StarsIn to be a table of that type. □

9.4.5 Exercises for Section 9.4

Exercise 9.4.1: Write type declarations for the following types:

a) NameType, with components for first, middle, and last names and a title.

* b) PersonType, with a name of the person and references to the persons that are their mother and father. You must use the type from part (a) in your declaration.

c) MarriageType, with the date of the marriage and references to the husband and wife.

Exercise 9.4.2: Redesign our running products database schema of Exercise 5.2.1 to use type declarations and reference attributes where appropriate. In particular, in the relations PC, Laptop, and Printer make the model attribute be a reference to the Product tuple for that model.

! **Exercise 9.4.3:** In Exercise 9.4.2 we suggested that model numbers in the tables PC, Laptop, and Printer could be references to tuples of the Product table. Is it also possible to make the model attribute in Product a reference to the tuple in the relation for that type of product? Why or why not?

* **Exercise 9.4.4:** Redesign our running battleships database schema of Exercise 5.2.4 to use type declarations and reference attributes where appropriate. The schema from Exercise 9.1.3 should suggest where reference attributes are useful. Look for many-one relationships and try to represent them using an attribute with a reference type.

9.5 Operations on Object-Relational Data

All appropriate SQL operations from previous chapters apply to tables that are declared with a UDT or that have attributes whose type is a UDT. There are also some entirely new operations we can use, such as reference-following. However, some familiar operations, especially those that access or modify columns whose type is a UDT, involve new syntax.

9.5.1 Following References

Suppose x is a value of type REF(T). Then x refers to some tuple t of type T. We can obtain tuple t itself, or components of t, by two means:

1. Operator -> has essentially the same meaning as this operator does in C. That is, if x is a reference to a tuple t, and a is an attribute of t, then x->a is the value of the attribute a in tuple t.
2. The DEREF operator applies to a reference and produces the tuple referenced.

Example 9.26: Let us use the relation StarsIn from Example 9.25 to find the movies in which Mel Gibson starred. Recall that the schema is

StarsIn(star, movie)

where star and movie are references to tuples of MovieStar and Movie, respectively. A possible query is:

```
1) SELECT DEREF(movie)
2) FROM StarsIn
3) WHERE star->name = 'Mel Gibson';
```

In line (3), the expression star->name produces the value of the name component of the MovieStar tuple referred to by the star component of any given StarsIn tuple. Thus, the WHERE clause identifies those StarsIn tuples whose star component are references to the Mel-Gibson MovieStar tuple. Line (1) then produces the movie tuple referred to by the movie component of those tuples. All three attributes — title, year, and inColor — will appear in the printed result.

Note that we could have replaced line (1) by:

```
1) SELECT movie
```

However, had we done so, we would have gotten a list of system-generated gibberish that serves as the internal unique identifiers for those tuples. We would not see the information in the referenced tuples. □

9.5.2 Accessing Attributes of Tuples with a UDT

When we define a relation to have a UDT, the tuples must be thought of as single objects, rather than lists with components corresponding to the attributes of the UDT. As a case in point, consider the relation Movie declared in Fig. 9.11. This relation has UDT MovieType, which has three attributes: title, year, and inColor. However, a tuple t in Movie has only *one* component, not three. That component is the object itself.

If we "drill down" into the object, we can extract the values of the three attributes in the type MovieType, as well as use any methods defined for that type. However, we have to access these attributes properly, since they are not attributes of the tuple itself. Rather, every UDT has an implicitly defined *observer method* for each attribute of that UDT. The name of the observer method for an attribute x is $x()$. We apply this method as we would any other method for this UDT; we attach it with a dot to an expression that evaluates to an object of this type. Thus, if t is a variable whose value is of type T, and x is an attribute of T, then $t.x()$ is the value of x in the tuple (object) denoted by t.

Example 9.27: Let us find, from the relation Movie of Fig. 9.11 the year(s) of movies with title *King Kong*. Here is one way to do so:

```
SELECT m.year()
FROM Movie m
WHERE m.title() = 'King Kong';
```


Even though the tuple variable m would appear not to be needed here, we need a variable whose value is an object of type MovieType — the UDT for relation Movie. The condition of the WHERE clause compares the constant 'King Kong' to the value of m.title(). The latter is the observer method for attribute title of type MovieType. Similarly, the value in the SELECT clause is expressed m.year(); this expression applies the observer method for year to the object m. □

9.5.3 Generator and Mutator Functions

In order to create data that conforms to a UDT, or to change components of objects with a UDT, we can use two kinds of methods that are created automatically, along with the observer methods, whenever a UDT is defined. These are:

1. A *generator method.* This method has the name of the type and no argument. It also has the unusual property that it may be invoked without being applied to any object. That is, if T is a UDT, then $T()$ returns an object of type T, with no values in its various components.

2. *Mutator methods.* For each attribute x of UDT T, there is a mutator method $x(v)$. When applied to an object of type T, it changes the x attribute of that object to have value v. Notice that the mutator and observer method for an attribute each have the name of the attribute, but differ in that the mutator has an argument.

Example 9.28: We shall write a PSM procedure that takes as arguments a street, a city, and a name, and inserts into the relation MovieStar (of type StarType according to Example 9.22) an object constructed from these values, using calls to the proper generator and mutator functions. Recall from Example 9.20 that objects of StarType have a name component that is a character string, but an address component that is itself an object of type AddressType. The procedure InsertStar is shown in Fig. 9.12.

Lines (2) through (4) introduce the arguments s, c, and n, which will provide values for a street, city, and star name, respectively. Lines (5) and (6) declare two local variables. Each is of one of the UDT's involved in the type for objects that exist in the relation MovieStar. At lines (7) and (8) we create empty objects of each of these two types.

Lines (9) and (10) put real values in the object newAddr; these values are taken from the procedure arguments that provide a street and a city. Line (11) similarly installs the argument n as the value of the name component in the object newStar. Then line (12) takes the entire newAddr object and makes it the value of the address component in newStar. Finally, line (13) inserts the constructed object into relation MovieStar. Notice that, as always, a relation that has a UDT as its type has but a single component, even if that component has several attributes, such as name and address in this example.

```
1)  CREATE PROCEDURE InsertStar(
2)      IN s CHAR(50),
3)      IN c CHAR(20),
4)      IN n CHAR(30)
    )
5)  DECLARE newAddr AddressType;
6)  DECLARE newStar StarType;

    BEGIN
7)      SET newAddr = AddressType();
8)      SET newStar = StarType();
9)      newAddr.street(s);
10)     newAddr.city(c);
11)     newStar.name(n);
12)     newStar.address(newAddr);
13)     INSERT INTO MovieStar VALUES(newStar);
    END;
```

Figure 9.12: Creating and storing a StarType object

To insert a star into MovieStar, we can call procedure InsertStar.

```
CALL InsertStar('345 Spruce St.', 'Glendale', 'Gwyneth Paltrow');
```

is an example. □

It is much simpler to insert objects into a relation with a UDT if your DBMS provides, or if you create, a generator function that takes values for the attributes of the UDT and returns a suitable object. For example, if we have functions AddressType(s,c) and StarType(n,a) that return objects of the indicated types, then we can make the insertion at the end of Example 9.28 with an INSERT statement of a familiar form:

```
INSERT INTO MovieStar VALUES(
    StarType('Gwyneth Paltrow',
        AddressType('345 Spruce St.', 'Glendale')));
```

9.5.4 Ordering Relationships on UDT's

Objects that are of some UDT are inherently abstract, in the sense that there is no way to compare two objects of the same UDT, either to test whether they are "equal" or whether one is less than another. Even two objects that have all components identical will not be considered equal unless we tell the system to regard them as equal. Similarly, there is no obvious way to sort the tuples of

a relation that has a UDT unless we define a function that tells which of two objects of that UDT precedes the other.

Yet there are many SQL operations that require either an equality test or both an equality and a "less than" test. For instance, we cannot eliminate duplicates if we can't tell whether two tuples are equal. We cannot group by an attribute whose type is a UDT unless there is an equality test for that UDT. We cannot use an ORDER BY clause or a comparison like $<$ in a WHERE clause unless we can compare any two elements.

To specify an ordering or comparison, SQL allows us to issue a CREATE ORDERING statement for any UDT. There are a number of forms this statement may take, and we shall only consider the two simplest options:

1. The statement

   CREATE ORDERING FOR T EQUALS ONLY BY STATE;

   says that two members of UDT T are considered equal if all of their corresponding components are equal. There is no $<$ defined on objects of UDT T.

2. The following statement

   CREATE ORDERING FOR T
   ORDERING FULL BY RELATIVE WITH F;

   says that any of the six comparisons (<, <=, >, >=, =, and <>) may be performed on objects of UDT T. To tell how objects x_1 and x_2 compare, we apply the function F to these objects. This function must be written so that $F(x_1, x_2) < 0$ whenever we want to conclude that $x_1 < x_2$; $F(x_1, x_2) = 0$ means that $x_1 = x_2$, and $F(x_1, x_2) > 0$ means that $x_1 > x_2$. If we replace "ORDERING FULL" with "EQUALS ONLY," then $F(x_1, x_2) = 0$ indicates that $x_1 = x_2$, while any other value of $F(x_1, x_2)$ means that $x_1 \neq x_2$. Comparison by $<$ is impossible in this case.

Example 9.29: Let us consider a possible ordering on the UDT StarType from Example 9.20. If we want only an equality on objects of this UDT, we could declare:

```
CREATE ORDERING FOR StarType EQUALS ONLY BY STATE;
```

That statement says that two objects of StarType are equal if and only if their names are the same as character strings, and their addresses are the same as objects of UDT AddressType.

The problem is that, unless we define an ordering for AddressType, an object of that type is not even equal to itself. Thus, we also need to create at least an equality test for AddressType. A simple way to do so is to declare that two AddressType objects are equal if and only if their streets and cities are each the same. We could do so by:

```
CREATE ORDERING FOR AddressType EQUALS ONLY BY STATE;
```

Alternatively, we could define a complete ordering of `AddressType` objects. One reasonable ordering is to order addresses first by cities, alphabetically, and among addresses in the same city, by street address, alphabetically. To do so, we have to define a function, say `AddrLEG`, that takes two `AddressType` arguments and returns a negative, zero, or positive value to indicate that the first is less than, equal to, or greater than the second. We declare:

```
CREATE ORDERING FOR AddressType
ORDER FULL BY RELATIVE WITH AddrLEG;
```

The function `AddrLEG` is shown in Fig. 9.13. Notice that if we reach line (7), it must be that the two `city` components are the same, so we compare the `street` components. Likewise, if we reach line (9), the only remaining possibility is that the cities are the same and the first street precedes the second alphabetically. □

```
1)  CREATE FUNCTION AddrLEG(
2)      x1 AddressType,
3)      x2 AddressType
4)  ) RETURNS INTEGER

5)  IF x1.city() < x2.city() THEN RETURN(-1)
6)  ELSEIF x1.city() > x2.city() THEN RETURN(1)
7)  ELSEIF x1.street() < x2.street() THEN RETURN(-1)
8)  ELSEIF x1.street() = x2.street() THEN RETURN(0)
9)  ELSE RETURN(1)
    END IF;
```

Figure 9.13: A comparison function for address objects

9.5.5 Exercises for Section 9.5

Exercise 9.5.1: Using the `StarsIn` relation of Example 9.25, and the `Movie` and `MovieStar` relations accessible through `StarsIn`, write the following queries:

* a) Find the names of the stars of *Ishtar*.

*! b) Find the titles and years of all movies in which at least one star lives in Malibu.

c) Find all the movies (objects of type `MovieType`) that starred Melanie Griffith.

! d) Find the movies (title and year) with at least five stars.

Exercise 9.5.2: Using your schema from Exercise 9.4.2, write the following queries. Don't forget to use references whenever appropriate.

a) Find the manufacturers of PC's with a hard disk larger than 60 gigabytes.

b) Find the manufacturers of laser printers.

! c) Produce a table giving for each model of laptop, the model of the laptop having the highest processor speed of any laptop made by the same manufacturer.

Exercise 9.5.3: Using your schema from Exercise 9.4.4, write the following queries. Don't forget to use references whenever appropriate and avoid joins (i.e., subqueries or more than one tuple variable in the `FROM` clause).

* a) Find the ships with a displacement of more than 35,000 tons.

b) Find the battles in which at least one ship was sunk.

! c) Find the classes that had ships launched after 1930.

!! d) Find the battles in which at least one US ship was damaged.

Exercise 9.5.4: Assuming the function `AddrLEG` of Fig. 9.13 is available, write a suitable function to compare objects of type `StarType`, and declare your function to be the basis of the ordering of `StarType` objects.

*! **Exercise 9.5.5:** Write a procedure to take a star name as argument and delete from `StarsIn` and `MovieStar` all tuples involving that star.

9.6 Summary of Chapter 9

- *Select-From-Where Statements in OQL*: OQL offers a select-from-where expression that resembles SQL's. In the `FROM` clause, we can declare variables that range over any collection, including both extents of classes (analogous to relations) and collections that are the values of attributes in objects.
- *Common OQL Operators*: OQL offers for-all, there-exists, `IN`, union, intersection, difference, and aggregation operators that are similar in spirit to SQL's. However, aggregation is always over a collection, not a column of a relation.
- *OQL Group-By*: OQL also offers a `GROUP BY` clause in select-from-where statements that is similar to SQL's. However, in OQL, the collection of objects in each group is explicitly accessible through a field name called `partition`.

- *Extracting Elements From OQL Collections*: We can obtain the lone member of a collection that is a singleton by applying the `ELEMENT` operator. The elements of a collection with more than one member can be accessed by first turning the collection into a list, using an `ORDER BY` clause in a select-from-where statement, and then using a loop in the surrounding host-language program to visit each element of the list in turn.
- *User-Defined Types in SQL:* Object-relational capabilities of SQL are centered around the UDT, or user-defined type. These types may be declared by listing their attributes and other information, as in table declarations. In addition, methods may be declared for UDT's.
- *Relations With a UDT as Type*: Instead of declaring the attributes of a relation, we may declare that relation to have a UDT. If we do so, then its tuples have one component, and this component is an object of the UDT.
- *Reference Types*: A type of an attribute can be a reference to a UDT. Such attributes essentially are pointers to objects of that UDT.
- *Object Identity for UDT's*: When we create a relation whose type is a UDT, we declare an attribute to serve as the "object-ID" of each tuple. This component is a reference to the tuple itself. Unlike in object-oriented systems, this "OID" column may be accessed by the user, although it is rarely meaningful.
- *Accessing components of a UDT*: SQL provides observer and mutator functions for each attribute of a UDT. These functions, respectively, return and change the value of that attribute when applied to any object of that UDT.

9.7 References for Chapter 9

The reference for OQL is the same as for ODL: [1]. Material on object-relational features of SQL can be obtained as described in the bibliographic notes to Chapter 6.

1. Cattell, R. G. G. (ed.), *The Object Database Standard: ODMG-99*, Morgan-Kaufmann, San Francisco, 1999.

Chapter 10

Logical Query Languages

Some query languages for the relational model resemble a logic more than they do the algebra that we introduced in Section 5.2. However, logic-based languages appear to be difficult for many programmers to grasp. Thus, we have delayed our coverage of logic until the end of our study of query languages.

We shall introduce Datalog, which is the simplest form of logic devised for the relational model. In its nonrecursive form, Datalog has the same power as the classical relational algebra. However, by allowing recursion, we can express queries in Datalog that cannot be expressed in SQL2 (except by adding procedural programming such as PSM). We discuss the complexities that come up when we allow recursive negation, and finally, we see how the solution provided by Datalog has been used to provide a way to allow meaningful recursion in the most recent SQL-99 standard.

10.1 A Logic for Relations

As an alternative to abstract query languages based on algebra, one can use a form of logic to express queries. The logical query language *Datalog* ("database logic") consists of if-then rules. Each of these rules expresses the idea that from certain combinations of tuples in certain relations we may infer that some other tuple is in some other relation, or in the answer to a query.

10.1.1 Predicates and Atoms

Relations are represented in Datalog by *predicates*. Each predicate takes a fixed number of arguments, and a predicate followed by its arguments is called an *atom*. The syntax of atoms is just like that of function calls in conventional programming languages; for example $P(x_1, x_2, \ldots, x_n)$ is an atom consisting of the predicate P with arguments $x_1, x_2, \ldots, x_n$.

In essence, a predicate is the name of a function that returns a boolean value. If R is a relation with n attributes in some fixed order, then we shall