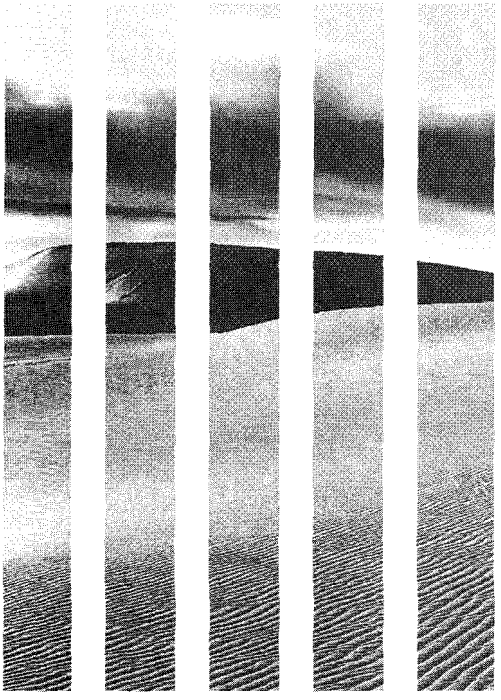


6

OBJECT AND OBJECT-RELATIONAL DATABASES

20

Concepts for Object Databases

In this chapter and the next, we discuss object-oriented data models and database systems.[1] Traditional data models and systems, such as relational, network, and hierarchical, have been quite successful in developing the database technology required for many traditional business database applications. However, they have certain shortcomings when more complex database applications must be designed and implemented—for example, databases for engineering design and manufacturing (CAD/CAM and CIM[2]), scientific experiments, telecommunications, geographic information systems, and multimedia.[3] These newer applications have requirements and characteristics that differ from those of traditional business applications, such as more complex structures for objects, longer-duration transactions, new data types for storing images or large textual items, and the need to define nonstandard application-specific operations. Object-oriented databases were proposed to meet the needs of these more complex applications. The object-oriented approach offers the flexibility to handle some of these requirements without

1. These databases are often referred to as **Object Databases** and the systems are referred to as **Object Database Management Systems** (ODBMS). However, because this chapter discusses many general object-oriented concepts, we will use the term *object-oriented* instead of just *object*.

2. Computer-Aided Design/Computer-Aided Manufacturing and Computer-Integrated Manufacturing.

3. Multimedia databases must store various types of multimedia objects, such as video, audio, images, graphics, and documents (see Chapter 24).

being limited by the data types and query languages available in traditional database systems. A key feature of object-oriented databases is the power they give the designer to specify both the *structure* of complex objects and the *operations* that can be applied to these objects.

Another reason for the creation of object-oriented databases is the increasing use of object-oriented programming languages in developing software applications. Databases are now becoming fundamental components in many software systems, and traditional databases were difficult to use with object-oriented software applications that are developed in an object-oriented programming language such as C++, SMALLTALK, or JAVA. Object-oriented databases are designed so they can be directly—or *seamlessly*—integrated with software that is developed using object-oriented programming languages.

The need for additional data modeling features has also been recognized by relational DBMS vendors, and newer versions of relational systems are incorporating many of the features that were proposed for object-oriented databases. This has led to systems that are characterized as *object-relational* or *extended relational* DBMSs (see Chapter 22). The latest version of the SQL standard for relational DBMSs includes some of these features.

Although many experimental prototypes and commercial object-oriented database systems have been created, they have not found widespread use because of the popularity of relational and object-relational systems. The experimental prototypes included the ORION system developed at MCC,[4] OPENOODB at Texas Instruments, the IRIS system at Hewlett-Packard laboratories, the ODE system at AT&T Bell Labs,[5] and the ENCORE/ObServer project at Brown University. Commercially available systems included GEMSTONE/OPAL of GemStone Systems, ONTOS of Ontos, Objectivity of Objectivity Inc., Versant of Versant Object Technology, ObjectStore of Object Design, ARDENT of ARDENT Software,[6] and POET of POET Software. These represent only a partial list of the experimental prototypes and commercial object-oriented database systems that were created.

As commercial object-oriented DBMSs became available, the need for a standard model and language was recognized. Because the formal procedure for approval of standards normally takes a number of years, a consortium of object-oriented DBMS vendors and users, called ODMG,[7] proposed a standard that is known as the ODMG-93 standard, which has since been revised. We will describe some features of the ODMG standard in Chapter 21.

Object-oriented databases have adopted many of the concepts that were developed originally for object-oriented programming languages.[8] In Section 20.1, we examine the origins of the object-oriented approach and discuss how it applies to database systems. Then, in Sections 20.2 through 20.6, we describe the key concepts utilized in many object-

4. Microelectronics and Computer Technology Corporation, Austin, Texas.

5. Now called Lucent Technologies.

6. Formerly O2 of O2 Technology.

7. Object Database Management Group.

8. Similar concepts were also developed in the fields of semantic data modeling and knowledge representation.

oriented database systems. Section 20.2 discusses *object identity, object structure*, and *type constructors*. Section 20.3 presents the concepts of *encapsulation of operations* and definition of *methods* as part of class declarations, and also discusses the mechanisms for storing objects in a database by making them *persistent*. Section 20.4 describes *type and class hierarchies* and *inheritance* in object-oriented databases, and Section 20.5 provides an overview of the issues that arise when *complex objects* need to be represented and stored. Section 20.6 discusses additional concepts, including *polymorphism, operator overloading, dynamic binding, multiple and selective inheritance*, and *versioning* and *configuration* of objects.

This chapter presents the general concepts of object-oriented databases, whereas Chapter 22 will present the ODMG standard. The reader may skip Sections 20.5 and 20.6 of this chapter if a less detailed introduction to the topic is desired.

20.1 Overview of Object-Oriented Concepts

This section gives a quick overview of the history and main concepts of object-oriented databases, or OODBs for short. The OODB concepts are then explained in more detail in Sections 20.2 through 20.6. The term *object-oriented*—abbreviated by OO or O-O—has its origins in OO programming languages, or OOPLs. Today OO concepts are applied in the areas of databases, software engineering, knowledge bases, artificial intelligence, and computer systems in general. OOPLs have their roots in the SIMULA language, which was proposed in the late 1960s. In SIMULA, the concept of a *class* groups together the internal data structure of an object in a class declaration. Subsequently, researchers proposed the concept of *abstract data type*, which hides the internal data structures and specifies all possible external operations that can be applied to an object, leading to the concept of *encapsulation*. The programming language SMALLTALK, developed at Xerox PARC[9] in the 1970s, was one of the first languages to explicitly incorporate additional OO concepts, such as message passing and inheritance. It is known as a *pure* OO programming language, meaning that it was explicitly designed to be object-oriented. This contrasts with *hybrid* OO programming languages, which incorporate OO concepts into an already existing language. An example of the latter is C++, which incorporates OO concepts into the popular C programming language.

An **object** typically has two components: state (value) and behavior (operations). Hence, it is somewhat similar to a *program variable* in a programming language, except that it will typically have a *complex data structure* as well as *specific operations* defined by the programmer.[10] Objects in an OOPL exist only during program execution and are hence called *transient objects*. An OO database can extend the existence of objects so that they are stored permanently, and hence the objects *persist* beyond program termination and can be retrieved later and shared by other programs. In other words, OO databases store

9. Palo Alto Research Center, Palo Alto, California.

10. Objects have many other characteristics, as we discuss in the rest of this chapter.

persistent objects permanently on secondary storage, and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as indexing mechanisms, concurrency control, and recovery. An OO database system interfaces with one or more OO programming languages to provide persistent and shared object capabilities.

One goal of OO databases is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, OO databases provide a unique system-generated *object identifier* (OID) for each object. We can compare this with the relational model where each relation must have a primary key attribute whose value identifies each tuple uniquely. In the relational model, if the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same object (for example, the object identifier may be represented as EMP_ID in one relation and as SSN in another).

Another feature of OO databases is that objects may have an *object structure* of *arbitrary complexity* in order to contain all of the necessary information that describes the object. In contrast, in traditional database systems, information about a complex object is often *scattered* over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.

The internal structure of an object in OOPLs includes the specification of **instance variables,** which hold the values that define the internal state of the object. Hence, an instance variable is similar to the concept of an *attribute* in the relational model, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some OO models insist that all operations a user can apply to an object must be predefined. This forces a *complete encapsulation* of objects. This rigid approach has been relaxed in most OO data models for several reasons. First, the database user often needs to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects. Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify on the fly.

To encourage encapsulation, an operation is defined in two parts. The first part, called the *signature* or *interface* of the operation, specifies the operation name and arguments (or parameters). The second part, called the *method* or *body*, specifies the *implementation* of the operation. Operations can be invoked by passing a *message* to an object, which includes the operation name and the parameters. The object then executes the method for that operation. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations. Hence, encapsulation provides a form of data and operation independence (see Chapter 2).

Another key concept in OO systems is that of type and class hierarchies and *inheritance*. This permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes. Hence, specification of object types can

proceed systematically. This makes it easier to develop the data types of a system incrementally, and to *reuse* existing type definitions when creating new types of objects.

One problem in early OO database systems involved representing *relationships* among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships, because it is useful to identify these relationships and make them visible to users. The ODMG standard has recognized this need and it explicitly represents binary relationships via a pair of *inverse references*—that is, by placing the OIDs of related objects within the objects themselves, and maintaining referential integrity, as we shall describe in Chapter 21.

Some OO systems provide capabilities for dealing with *multiple versions* of the same object—a feature that is essential in design and engineering applications. For example, an old version of an object that represents a tested and verified design should be retained until the new version is tested and verified. A new version of a complex object may include only a few new versions of its component objects, whereas other components remain unchanged. In addition to permitting versioning, OO databases should also allow for *schema evolution*, which occurs when type declarations are changed or when new types or relationships are created. These two features are not specific to OODBs and should ideally be included in all types of DBMSs.[11]

Another OO concept is *operator overloading*, which refers to an operation's ability to be applied to different types of objects; in such a situation, an *operation name* may refer to several distinct *implementations*, depending on the type of objects it is applied to. This feature is also called *operator polymorphism*. For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle. This may require the use of *late binding* of the operation name to the appropriate method at run-time, when the type of object to which the operation is applied becomes known.

This section provided an overview of the main concepts of OO databases. In Sections 20.2 through 20.6, we discuss these concepts in more detail.

20.2 Object Identity, Object Structure, and Type Constructors

In this section we first discuss the concept of object identity, and then we present the typical structuring operations for defining the structure of the state of an object. These structuring operations are often called **type constructors.** They define basic data-structuring operations that can be combined to form complex object structures.

11. Several schema evolution operations, such as ALTER TABLE, are already defined in the relational SQL standard (see Section 8.3).

20.2.1 Object Identity

An OO database system provides a **unique identity** to each independent object stored in the database. This unique identity is typically implemented via a unique, system-generated **object identifier,** or **OID.** The value of an OID is not visible to the external user, but it is used internally by the system to identify each object uniquely and to create and manage inter-object references. The OID can be assigned to program variables of the appropriate type when needed.

The main property required of an OID is that it be **immutable;** that is, the OID value of a particular object should not change. This preserves the identity of the real-world object being represented. Hence, an OO database system must have some mechanism for generating OIDs and preserving the immutability property. It is also desirable that each OID be used only once; that is, even if an object is removed from the database, its OID should not be assigned to another object. These two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed or corrected. It is also generally considered inappropriate to base the OID on the physical address of the object in storage, since the physical address can change after a physical reorganization of the database. However, some systems do use the physical address as OID to increase the efficiency of object retrieval. If the physical address of the object changes, an *indirect pointer* can be placed at the former address, which gives the new physical location of the object. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the current physical address of the object in storage.

Some early OO data models required that everything—from a simple value to a complex object—be represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. This allows two basic values to have different OIDs, which can be useful in some cases. For example, the integer value 50 can be used sometimes to mean a weight in kilograms and at other times to mean the age of a person. Then, two basic objects with distinct OIDs could be created, but both objects would represent the integer value 50. Although useful as a theoretical model, this is not very practical, since it may lead to the generation of too many OIDs. Hence, most OO database systems allow for the representation of both objects and **values.** Every object must have an immutable OID, whereas a value has no OID and just stands for itself. Hence, a value is typically stored within an object and *cannot be referenced* from other objects. In some systems, complex structured values can also be created without having a corresponding OID if needed.

20.2.2 Object Structure

In OO databases, the state (current value) of a complex object may be constructed from other objects (or other values) by using certain **type constructors.** One formal way of representing such objects is to view each object as a triple (i, c, v), where i is a unique *object identifier* (the OID), c is a *type constructor*[12] (that is, an indication of how the object state is

12. This is different from the constructor operation that is used in C++ and other OOPLs to create new objects.

constructed), and v is the object state (or *current value*). The data model will typically include several type constructors. The three most basic constructors are **atom, tuple,** and **set.** Other commonly used constructors include **list, bag,** and **array.** The atom constructor is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly.

The object state v of an object (i, c, v) is interpreted based on the constructor c. If c = atom, the state (value) v is an atomic value from the domain of basic values supported by the system. If c = set, the state v is a *set of object identifiers* $\{i_1, i_2, \ldots, i_n\}$, which are the OIDs for a set of objects that are typically of the same type. If c = tuple, the state v is a tuple of the form $<a_1:i_1, a_2:i_2, \ldots, a_n:i_n>$, where each a_j is an attribute name[13] and each i_j is an OID. If c = list, the value v is an *ordered list* $[i_1, i_2, \ldots, i_n]$ of OIDs of objects of the same type. A list is similar to a set except that the OIDs in a list are *ordered*, and hence we can refer to the first, second, or j^{th} object in a list. For c = array, the state of the object is a single-dimensional array of object identifiers. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size. The difference between *set* and *bag*[14] is that all elements in a set must be distinct whereas a bag can have duplicate elements.

This model of objects allows arbitrary nesting of the set, list, tuple, and other constructors. The state of an object that is not of type atom will refer to other objects by their object identifiers. Hence, the only case where an actual value appears is in *the state of an object of type atom.*[15]

The type constructors **set, list, array,** and **bag** are called **collection types** (or **bulk types**), to distinguish them from basic types and tuple types. The main characteristic of a collection type is that the state of the object will be a *collection of objects* that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The **tuple** type constructor is often called a **structured type,** since it corresponds to the **struct** construct in the C and C++ programming languages.

EXAMPLE 1: A Complex Object

We now represent some objects from the relational database shown in Figure 5.6, using the preceding model, where an object is defined by a triple (OID, type constructor, state) and the available type constuctors are atom, set, and tuple. We use $i_1, i_2, i_3, \ldots$ to stand for unique system-generated object identifiers. Consider the following objects:

$o_1 = (i_1,$ atom, 'Houston')

$o_2 = (i_2,$ atom, 'Bellaire')

$o_3 = (i_3,$ atom, 'Sugarland')

13. Also called an *instance variable name* in OO terminology.

14. Also called a multiset.

15. As we noted earlier, it is not practical to generate a unique system identifier for every value, so real systems allow for both OIDs and *structured value*, which can be structured by using the same type constructors as objects, except that a value *does not have* an OID.

o_4 = (i_4, atom, 5)

o_5 = (i_5, atom, 'Research')

o_6 = (i_6, atom, '1988-05-22')

o_7 = (i_7, set, {i_1, i_2, i_3})

o_8 = (i_8, tuple, <DNAME:i_5, DNUMBER:i_4, MGR:i_9, LOCATIONS:i_7, EMPLOYEES:i_{10}, PROJECTS:i_{11}>)

o_9 = (i_9, tuple, <MANAGER:i_{12}, MANAGER_START_DATE:i_6>)

o_{10} = (i_{10}, set, {i_{12}, i_{13}, i_{14}})

o_{11} = (i_{11}, set {i_{15}, i_{16}, i_{17}})

o_{12} = (i_{12}, tuple, <FNAME:i_{18}, MINIT:i_{19}, LNAME:i_{20}, SSN:i_{21}, . . . , SALARY:i_{26}, SUPERVISOR:i_{27}, DEPT:i_8>)

. . .

The first six objects (o_1–o_6) listed here represent atomic values. There will be many similar objects, one for each distinct constant atomic value in the database.[16] Object o_7 is a set-valued object that represents the set of locations for department 5; the set {i_1, i_2, i_3} refers to the atomic objects with values {'Houston', 'Bellaire', 'Sugarland'}. Object o_8 is a tuple-valued object that represents department 5 itself, and has the attributes DNAME, DNUMBER, MGR, LOCATIONS, and so on. The first two attributes DNAME and DNUMBER have atomic objects o_5 and o_4 as their values. The MGR attribute has a tuple object o_9 as its value, which in turn has two attributes. The value of the MANAGER attribute is the object whose OID is i_{12}, which represents the employee 'John B. Smith' who manages the department, whereas the value of MANAGER_START_DATE is another atomic object whose value is a date. The value of the EMPLOYEES attribute of o_8 is a set object with OID = i_{10}, whose value is the set of object identifiers for the employees who work for the DEPARTMENT (objects i_{12}, plus i_{13} and i_{14}, which are not shown). Similarly, the value of the PROJECTS attribute of o_8 is a set object with OID = i_{11}, whose value is the set of object identifiers for the projects that are controlled by department number 5 (objects i_{15}, i_{16}, and i_{17}, which are not shown). The object whose OID = i_{12} represents the employee 'John B. Smith' with all its atomic attributes (FNAME, MINIT, LNAME, SSN, . . ., SALARY, that are referencing the atomic objects i_{18}, i_{19}, i_{20}, i_{21}, . . ., i_{26}, respectively (not shown)) plus SUPERVISOR which references the employee object with OID = i_{27} (this represents 'James E. Borg' who supervises 'John B. Smith' but is not shown) and DEPT which references the department object with OID = i_8 (this represents department number 5 where 'John B. Smith' works).

In this model, an object can be represented as a graph structure that can be constructed by recursively applying the type constructors. The graph representing an object o_i can be constructed by first creating a node for the object o_i itself. The node for o_i is labeled with the OID and the object constructor c. We also create a node in the graph for each basic atomic

16. These atomic objects are the ones that may cause a problem, due to the use of too many object identifiers, if this model is implemented directly.

value. If an object o_i has an atomic value, we draw a directed arc from the node representing o_i to the node representing its basic value. If the object value is constructed, we draw directed arcs from the object node to a node that represents the constructed value. Figure 20.1 shows the graph for the example DEPARTMENT object o_8 given earlier.

The preceding model permits two types of definitions in a comparison of the *states of two objects* for equality. Two objects are said to have **identical states** (deep equality) if the graphs representing their states are identical in every respect, including the OIDs at every level. Another, weaker definition of equality is when two objects have **equal states** (shallow equality). In this case, the graph structures must be the same, and all the corresponding atomic values in the graphs should also be the same. However, some corresponding internal nodes in the two graphs may have objects with *different* OIDs.

EXAMPLE 2: Identical Versus Equal Objects

A example can illustrate the difference between the two definitions for comparing object states for equality. Consider the following objects o_1, o_2, o_3, o_4, o_5, and o_6:

$o_1 = (i_1, \text{tuple}, <a_1:i_4, a_2:i_6>)$

$o_2 = (i_2, \text{tuple}, <a_1:i_5, a_2:i_6>)$

$o_3 = (i_3, \text{tuple}, <a_1:i_4, a_2:i_6>)$

$o_4 = (i_4, \text{atom}, 10)$

$o_5 = (i_5, \text{atom}, 10)$

$o_6 = (i_6, \text{atom}, 20)$

The objects o_1 and o_2 have *equal* states, since their states at the atomic level are the same but the values are reached through distinct objects o_4 and o_5. However, the states of objects o_1 and o_3 are *identical*, even though the objects themselves are not because they have distinct OIDs. Similarly, although the states of o_4 and o_5 are identical, the actual objects o_4 and o_5 are equal but not identical, because they have distinct OIDs.

20.2.3 Type Constructors

An **object definition language** (ODL)[17] that incorporates the preceding type constructors can be used to define the object types for a particular database application. In Chapter 21, we shall describe the standard ODL of ODMG, but we first introduce the concepts gradually in this section using a simpler notation. The type constructors can be used to define the *data structures* for an OO *database schema*. In Section 20.3 we will see how to incorporate the definition of *operations* (or methods) into the OO schema. Figure 20.2 shows how we may declare Employee and Department types corresponding to the object instances shown

17. This would correspond to the DDL (Data Definition Language) of the database system (see Chapter 2).

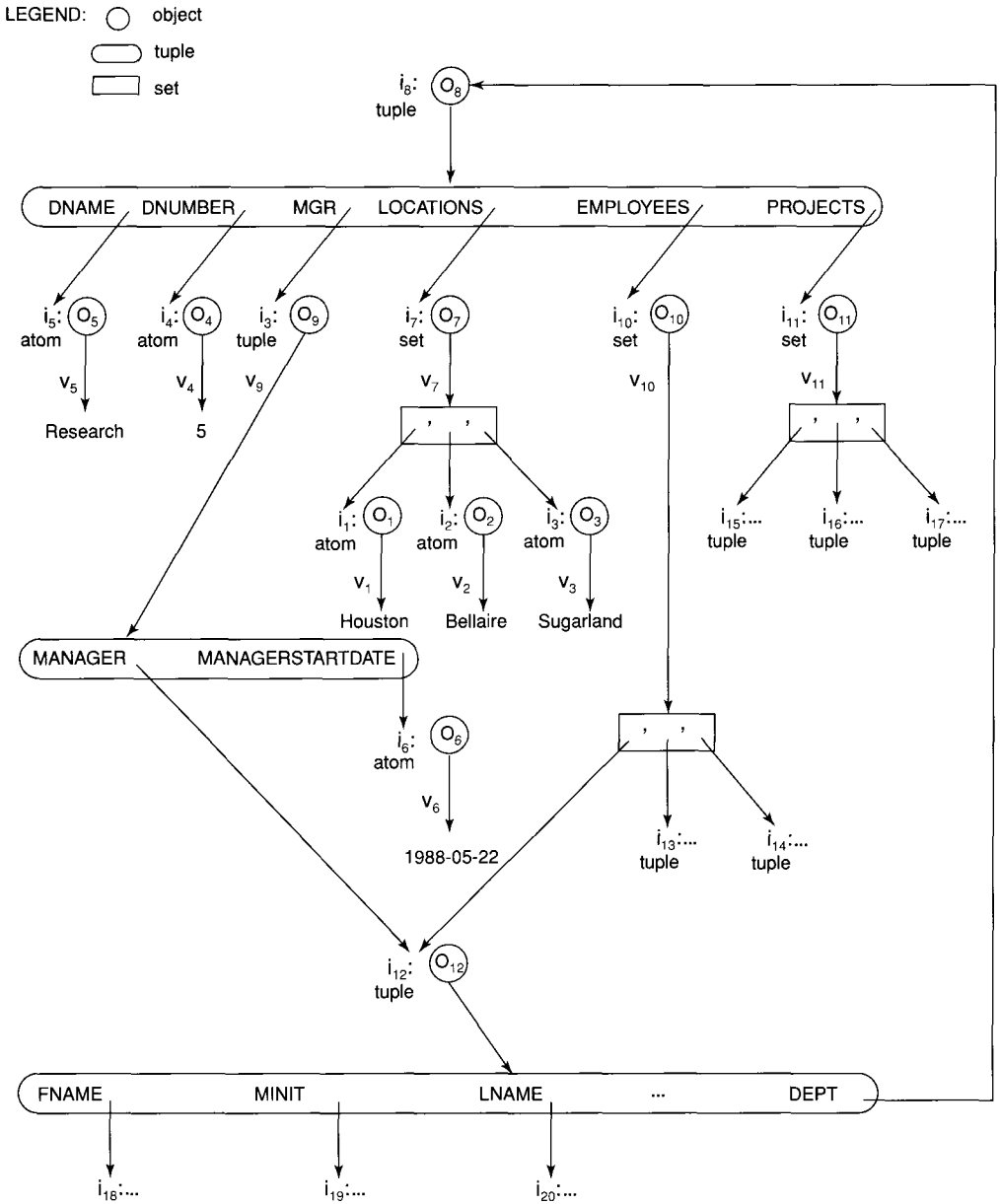


FIGURE 20.1 Representation of a DEPARTMENT complex object as a graph.

in Figure 20.1. In Figure 20.2, the Date type is defined as a tuple rather than an atomic value as in Figure 20.1. We use the keywords tuple, set, and list for the type constructors, and the available standard data types (integer, string, float, and so on) for atomic types.

```
define type Employee:
   tuple (    fname:        string;
              minit:        char;
              lname:        string;
              ssn:          string;
              birthdate:    Date;
              address:      string;
              sex:          char;
              salary:       float;
              supervisor:   Employee;
              dept:         Department;      );
define type Date
   tuple (    year:         integer;
              month:        integer;
              day:          integer;    );
define type Department
   tuple (    dname:        string;
              dnumber:      integer;
              mgr:          tuple (    manager:    Employee;
                                       startdate:  Date;       );
              locations:    set(string);
              employees:    set(Employee);
              projects      set(Project);  );
```

FIGURE 20.2 Specifying the object types Employee, Date, and Department using type constructors.

Attributes that refer to other objects—such as dept of Employee or projects of Department—are basically **references** to other objects and hence serve to represent *relationships* among the object types. For example, the attribute dept of Employee is of type Department, and hence is used to refer to a specific Department object (where the Employee works). The value of such an attribute would be an OID for a specific Department object. A binary relationship can be represented in one direction, or it can have an *inverse reference*. The latter representation makes it easy to traverse the relationship in both directions. For example, the attribute employees of Department has as its value a *set of references* (that is, a set of OIDs) to objects of type Employee; these are the employees who work for the department. The inverse is the reference attribute dept of Employee. We will see in Chapter 21 how the ODMG standard allows inverses to be explicitly declared as relationship attributes to ensure that inverse references are consistent.

20.3 Encapsulation of Operations, Methods, and Persistence

The concept of *encapsulation* is one of the main characteristics of OO languages and systems. It is also related to the concepts of *abstract data types* and *information hiding* in programming languages. In traditional database models and systems, this concept was not

applied, since it is customary to make the structure of database objects visible to users and external programs. In these traditional models, a number of standard database operations are applicable to objects of all types. For example, in the relational model, the operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to *any relation* in the database. The relation and its attributes are visible to users and to external programs that access the relation by using these operations.

20.3.1 Specifying Object Behavior via Class Operations

The concepts of information hiding and encapsulation can be applied to database objects. The main idea is to define the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type. The internal structure of the object is hidden, and the object is accessible only through a number of predefined operations. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations. Still other operations may perform a combination of retrieval, calculation, and update. In general, the **implementation** of an operation can be specified in a *general-purpose programming language* that provides flexibility and power in defining the operations.

The external users of the object are only made aware of the **interface** of the object type, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of the internal data structures of the object and the implementation of the operations that access these structures. In OO terminology, the interface part of each operation is called the **signature,** and the operation implementation is called a **method.** Typically, a method is invoked by sending a **message** to the object to execute the corresponding method. Notice that, as part of executing a method, a subsequent message to another object may be sent, and this mechanism may be used to return values from the objects to the external environment or to other objects.

For database applications, the requirement that all objects be completely encapsulated is too stringent. One way of relaxing this requirement is to divide the structure of an object into **visible** and **hidden** attributes (instance variables). Visible attributes may be directly accessed for reading by external operators, or by a high-level query language. The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations. Most OODBMSs employ high-level query languages for accessing visible attributes. In Chapter 21, we will describe the OQL query language that is proposed as a standard query language for OODBs.

In most cases, operations that *update* the state of an object are encapsulated. This is a way of defining the update semantics of the objects, given that in many OO data models, few integrity constraints are predefined in the schema. Each type of object has its integrity constraints *programmed into the methods* that create, delete, and update the objects by explicitly writing code to check for constraint violations and to handle exceptions. In such cases, all update operations are implemented by encapsulated operations. More recently, the ODL for the ODMG standard allows the specification of some common

constraints such as keys and inverse relationships (referential integrity) so that the system can automatically enforce these constraints (see Chapter 21).

The term **class** is often used to refer to an object type definition, along with the definitions of the operations for that type.[18] Figure 20.3 shows how the type definitions of Figure 20.2 may be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition. A method (implementation) for each operation must be defined elsewhere, using a programming language. Typical operations include the **object constructor** operation, which is used to create a new object, and the **destructor** operation, which is used to destroy an object. A number of **object modifier** operations can

```
define class Employee:
   type tuple(  fname:              string;
                minit:              char;
                lname:              string;
                ssn:                string;
                birthdate:          Date;
                address:            string;
                sex:                char;
                salary:             float;
                supervisor:         Employee;
                dept:               Department;          );
   operations   age:                integer;
                create_emp:         Employee;
                destroy_emp:        boolean;
end Employee;

define class Department
   type tuple(  dname:              string;
                dnumber:            integer;
                mgr:                tuple (    manager: Employee;
                                               startdate: Date;          );
                locations:          set(string);
                employees:          set(Employee);
                projects:           set(Project);      );
   operations   no_of_emps:         integer;
                create_dept:        Department;
                destroy_dept:       boolean;
                assign_emp(e: Employee): boolean;
          (* adds an employee to the department *)
                remove_emp(e: Employee): boolean;
          (* removes an employee from the department *)
end Department;
```

FIGURE 20.3 Adding operations to the definitions of Employee and Department.

18. This definition of *class* is similar to how it is used in the popular C++ programming language. The ODMG standard uses the word *interface* in addition to *class* (see Chapter 21). In the EER model, the term *class* was used to refer to an object type, along with the set of all objects of that type (see Chapter 4).

also be declared to modify the states (values) of various attributes of an object. Additional operations can **retrieve** information about the object.

An operation is typically applied to an object by using the **dot notation.** For example, if d is a reference to a department object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted). The only exception is the constructor operation, which returns a reference to a new Department object. Hence, it is customary to have a default name for the constructor operation that is the name of the class itself, although this was not used in Figure 20.3.[19] The dot notation is also used to refer to attributes of an object—for example, by writing d.dnumber or d.mgr.startdate.

20.3.2 Specifying Object Persistence via Naming and Reachability

An OODBMS is often closely coupled with an OOPL. The OOPL is used to specify the method implementations as well as other application code. An object is typically created by some executing application program, by invoking the object constructor operation. Not all objects are meant to be stored permanently in the database. **Transient objects** exist in the executing program and disappear once the program terminates. **Persistent objects** are stored in the database and persist after program termination. The typical mechanisms for making an object persistent are *naming* and *reachability*.

The **naming mechanism** involves giving an object a unique persistent name through which it can be retrieved by this and other programs. This persistent object name can be given via a specific statement or operation in the program, as illustrated in Figure 20.4. All such names given to objects must be unique within a particular database. Hence, the named persistent objects are used as **entry points** to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called **reachability.** The reachability mechanism works by making the object reachable from some persistent object. An object B is said to be **reachable** from an object A if a sequence of references in the object graph lead from object A to object B. For example, all the objects in Figure 20.1 are reachable from object o_8; hence, if o_8 is made persistent, all the other objects in Figure 20.1 also become persistent.

If we first create a named persistent object N, whose state is a *set* or *list* of objects of some class C, we can make objects of C persistent by *adding them* to the set or list, and thus making them reachable from N. Hence, N defines a **persistent collection** of objects of class C. For example, we can define a class DepartmentSet (see Figure 20.4) whose objects are of type **set**(Department).[20] Suppose that an object of type DepartmentSet is

19. Default names for the constructor and destructor operations exist in the C++ programming language. For example, for class Employee, the *default constructor name* is Employee and the *default destructor name* is ~Employee. It is also common to use the *new* operation to create new objects.

20. As we shall see in Chapter 21, the ODMG ODL syntax uses **set**<Department> instead of **set**(Department).

```
define class DepartmentSet:
   type        set(Department);
   operations  add_dept(d: Department): boolean;
          (* adds a department to the DepartmentSet object *)
               remove_dept(d: Department): boolean;
          (* removes a department from the DepartmentSet object *)
               create_dept_set:      DepartmentSet;
               destroy_dept_set:     boolean;
end DepartmentSet;

   ...

persistent name AllDepartments: DepartmentSet;
(* AllDepartments is a persistent named object of type DepartmentSet *)

   ...

d:= create_dept;
(* create a new Department object in the variable d *)

   ...

b:= AllDepartments.add_dept(d);
(* make d persistent by adding it to the persistent set AllDepartments *)

   ...
```

FIGURE 20.4 Creating persistent objects by naming and reachability.

created, and suppose that it is named AllDepartments and thus made persistent, as illustrated in Figure 20.4. Any Department object that is added to the set of AllDepartments by using the add_dept operation becomes persistent by virtue of its being reachable from AllDepartments. The AllDepartments object is often called the **extent** of the class Department, as it will hold all persistent objects of type Department. As we shall see in Chapter 21, the ODMG ODL standard gives the schema designer the option of naming an extent as part of class definition.

Notice the difference between traditional database models and OO databases in this respect. In traditional database models, such as the relational model or the EER model, *all* objects are assumed to be persistent. Hence, when an entity type or class such as EMPLOYEE is defined in the EER model, it represents both the *type declaration* for EMPLOYEE and a *persistent set* of *all* EMPLOYEE objects. In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) or list(EMPLOYEE) whose value is the *collection of references* to all persistent EMPLOYEE objects, if this is desired, as illustrated in Figure 20.4.[21] This allows transient and persistent objects to follow the same type and class declarations of the ODL and the OOPL. In general, it is possible to define several persistent collections for the same class definition, if desired.

21. Some systems, such as POET, automatically create the extent for a class.

20.4 Type and Class Hierarchies and Inheritance

Another main characteristic of OO database systems is that they allow type hierarchies and inheritance. Type hierarchies in databases usually imply a constraint on the extents corresponding to the types in the hierarchy. We first discuss type hierarchies (in Section 20.4.1), and then the constraints on the extents (in Section 20.4.2). We use a different OO model in this section—a model in which attributes and operations are treated uniformly—since both attributes and operations can be inherited. In Chapter 21, we will discuss the inheritance model of the ODMG standard, which differs from the model discussed here.

20.4.1 Type Hierarchies and Inheritance

In most database applications, there are numerous objects of the same type or class. Hence, OO databases must provide a capability for classifying objects based on their type, as do other database systems. But in OO databases, a further requirement is that the system permit the definition of new types based on other predefined types, leading to a **type (or class) hierarchy.**

Typically, a type is defined by assigning it a type name and then defining a number of attributes (instance variables) and operations (methods) for the type.[22] In some cases, the attributes and operations are together called *functions*, since attributes resemble functions with zero arguments. A function name can be used to refer to the value of an attribute or to refer to the resulting value of an operation (method). In this section, we use the term **function** to refer to both attributes *and* operations of an object type, since they are treated similarly in a basic introduction to inheritance.[23]

A type in its simplest form can be defined by giving it a **type name** and then listing the names of its visible (*public*) **functions.** When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:

```
TYPE_NAME: function, function, ..., function
```

For example, a type that describes characteristics of a PERSON may be defined as follows:

```
PERSON: Name, Address, Birthdate, Age, SSN
```

In the PERSON type, the Name, Address, SSN, and Birthdate functions can be implemented as stored attributes, whereas the Age function can be implemented as a method that calculates the Age from the value of the Birthdate attribute and the current date.

---

22. In this section, we will use the terms *type* and *class* as meaning the same thing—namely, the attributes *and* operations of some type of object.

23. We will see in Chapter 21 that types with functions are similar to the interfaces used in ODMG ODL.

The concept of **subtype** is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which we shall call the **supertype.** For example, suppose that we want to define two new types EMPLOYEE and STUDENT as follows:

```
EMPLOYEE: Name, Address, Birthdate, Age, SSN, Salary, HireDate, Seniority
STUDENT: Name, Address, Birthdate, Age, SSN, Major, GPA
```

Since both STUDENT and EMPLOYEE include all the functions defined for PERSON plus some additional functions of their own, we can declare them to be **subtypes** of PERSON. Each will inherit the previously defined functions of PERSON—namely, Name, Address, Birthdate, Age, and SSN. For STUDENT, it is only necessary to define the new (local) functions Major and GPA, which are not inherited. Presumably, Major can be defined as a stored attribute, whereas GPA may be implemented as a method that calculates the student's grade point average by accessing the Grade values that are internally stored (hidden) within each STUDENT object as *private attributes*. For EMPLOYEE, the Salary and HireDate functions may be stored attributes, whereas Seniority may be a method that calculates Seniority from the value of HireDate.

The idea of defining a type involves defining all of its functions and implementing them either as attributes or as methods. When a subtype is defined, it can then inherit all of these functions and their implementations. Only functions that are specific or **local** to the subtype, and hence are not specified in the supertype, need to be defined and implemented. Therefore, we can declare EMPLOYEE and STUDENT as follows:

```
EMPLOYEE subtype-of PERSON: Salary, HireDate, Seniority
STUDENT subtype-of PERSON: Major, GPA
```

In general, a subtype includes *all* of the functions that are defined for its supertype plus some additional functions that are specific only to the subtype. Hence, it is possible to generate a **type hierarchy** to show the supertype/subtype relationships among all the types declared in the system.

As another example, consider a type that describes objects in plane geometry, which may be defined as follows:

```
GEOMETRY_OBJECT: Shape, Area, ReferencePoint
```

For the GEOMETRY_OBJECT type, Shape is implemented as an attribute (its domain can be an enumerated type with values 'triangle', 'rectangle', 'circle', and so on), and Area is a method that is applied to calculate the area. ReferencePoint specifies the coordinates of a point that determines the object location. Now suppose that we want to define a number of subtypes for the GEOMETRY_OBJECT type, as follows:

```
RECTANGLE subtype-of GEOMETRY_OBJECT: Width, Height
TRIANGLE subtype-of GEOMETRY_OBJECT: Side1, Side2, Angle
CIRCLE subtype-of GEOMETRY_OBJECT: Radius
```

Notice that the Area operation may be implemented by a different method for each subtype, since the procedure for area calculation is different for rectangles,

triangles, and circles. Similarly, the attribute ReferencePoint may have a different meaning for each subtype; it might be the center point for RECTANGLE and CIRCLE objects, and the vertex point between the two given sides for a TRIANGLE object. Some OO database systems allow the **renaming** of inherited functions in different subtypes to reflect the meaning more closely.

An alternative way of declaring these three subtypes is to specify the value of the Shape attribute as a condition that must be satisfied for objects of each subtype:

```
RECTANGLE subtype-of GEOMETRY_OBJECT (Shape='rectangle'): Width,
Height
TRIANGLE subtype-of GEOMETRY_OBJECT (Shape='triangle'): Side1, Side2,
Angle
CIRCLE subtype-of GEOMETRY_OBJECT (Shape='circle'): Radius
```

Here, only GEOMETRY_OBJECT objects whose Shape='rectangle' are of the subtype RECTANGLE, and similarly for the other two subtypes. In this case, all functions of the GEOMETRY_OBJECT supertype are inherited by each of the three subtypes, but the value of the Shape attribute is restricted to a specific value for each.

Notice that type definitions describe objects but *do not* generate objects on their own. They are just declarations of certain types; and as part of that declaration, the implementation of the functions of each type is specified. In a database application, there are many objects of each type. When an object is created, it typically belongs to one or more of these types that have been declared. For example, a circle object is of type CIRCLE and GEOMETRY_OBJECT (by inheritance). Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are meaningful to the database application.

20.4.2 Constraints on Extents Corresponding to a Type Hierarchy[24]

In most OO databases, the collection of objects in an extent has the same type or class. However, this is not a necessary condition. For example, SMALLTALK, a so-called *typeless* OO language, allows a collection of objects to contain objects of different types. This can also be the case when other non-object-oriented typeless languages, such as LISP, are extended with OO concepts. However, since the majority of OO databases support types, we will assume that **extents** are collections of objects of the same type for the remainder of this section.

It is common in database applications that each type or subtype will have an extent associated with it, which holds the collection of all persistent objects of that type or subtype. In this case, the constraint is that every object in an extent that corresponds to a subtype must also be a member of the *extent* that corresponds to its supertype. Some OO database systems have a predefined system type (called the ROOT class or the OBJECT class)

24. In the second edition of this book, we used the title *Class Hierarchies* to describe these extent constraints. Because the word *class* has too many different meanings, *extent* is used in this edition. This is also more consistent with ODMG terminology (see Chapter 21).

whose extent contains all the objects in the system.[25] Classification then proceeds by assigning objects into additional subtypes that are meaningful to the application, creating a **type hierarchy** or **class hierarchy** for the system. All extents for system- and user-defined classes are subsets of the extent corresponding to the class OBJECT, directly or indirectly. In the ODMG model (see Chapter 21), the user may or may not specify an extent for each class (type), depending on the application.

In most OO systems, a distinction is made between persistent and transient objects and collections. A **persistent collection** holds a collection of objects that is stored permanently in the database and hence can be accessed and shared by multiple programs. A **transient collection** exists temporarily during the execution of a program but is not kept when the program terminates. For example, a transient collection may be created in a program to hold the result of a query that selects some objects from a persistent collection and copies those objects into the transient collection. The transient collection holds the same type of objects as the persistent collection. The program can then manipulate the objects in the transient collection, and once the program terminates, the transient collection ceases to exist. In general, numerous collections—transient or persistent—may contain objects of the same type.

Notice that the type constructors discussed in Section 20.2 permit the state of one object to be a collection of objects. Hence, collection objects whose types are based on the *set constructor* can define a number of collections—one corresponding to each object. The set-valued objects themselves are members of another collection. This allows for multilevel classification schemes, where an object in one collection has as its state a collection of objects of a different class.

As we shall see in Chapter 21, the ODMG model distinguishes between type inheritance—called interface inheritance and denoted by the ":" symbol—and the extent inheritance constraint—denoted by the keyword EXTEND.

20.5 Complex Objects

A principal motivation that led to the development of OO systems was the desire to represent complex objects. There are two main types of complex objects: structured and unstructured. A structured complex object is made up of components and is defined by applying the available type constructors recursively at various levels. An unstructured complex object typically is a data type that requires a large amount of storage, such as a data type that represents an image or a large textual object.

20.5.1 Unstructured Complex Objects and Type Extensibility

An **unstructured complex object** facility provided by a DBMS permits the storage and retrieval of large objects that are needed by the database application. Typical examples of

25. This is called OBJECT in the ODMG model (see Chapter 21).

such objects are *bitmap images* and *long text strings* (such as documents); they are also known as **binary large objects,** or **BLOBs** for short. Character strings are also known as **character large objects,** or **CLOBs** for short. These objects are unstructured in the sense that the DBMS does not know what their structure is—only the application that uses them can interpret their meaning. For example, the application may have functions to display an image or to search for certain keywords in a long text string. The objects are considered complex because they require a large area of storage and are not part of the standard data types provided by traditional DBMSs. Because the object size is quite large, a DBMS may retrieve a portion of the object and provide it to the application program before the whole object is retrieved. The DBMS may also use buffering and caching techniques to prefetch portions of the object before the application program needs to access them.

The DBMS software does not have the capability to directly process selection conditions and other operations based on values of these objects, unless the application provides the code to do the comparison operations needed for the selection. In an OODBMS, this can be accomplished by defining a new abstract data type for the uninterpreted objects and by providing the methods for selecting, comparing, and displaying such objects. For example, consider objects that are two-dimensional bitmap images. Suppose that the application needs to select from a collection of such objects only those that include a certain pattern. In this case, the user must provide the pattern recognition program as a method on objects of the bitmap type. The OODBMS then retrieves an object from the database and runs the method for pattern recognition on it to determine whether the object includes the required pattern.

Because an OODBMS allows users to create new types, and because a type includes both structure and operations, we can view an OODBMS as having an **extensible type system.** We can create libraries of new types by defining their structure and operations, including complex types. Applications can then use or modify these types, in the latter case by creating subtypes of the types provided in the libraries. However, the DBMS internals must provide the underlying storage and retrieval capabilities for objects that require large amounts of storage so that the operations may be applied efficiently. Many OODBMSs provide for the storage and retrieval of large unstructured objects such as character strings or bit strings, which can be passed "as is" to the application program for interpretation. Recently, relational and extended relational DBMSs have also been able to provide such capabilities. Special indexing techniques are also being developed.

20.5.2 Structured Complex Objects

A **structured complex object** differs from an unstructured complex object in that the object's structure is defined by repeated application of the type constructors provided by the OODBMS. Hence, the object structure is defined and known to the OODBMS. As an example, consider the DEPARTMENT object shown in Figure 20.1. At the first level, the object has a tuple structure with six attributes: DNAME, DNUMBER, MGR, LOCATIONS, EMPLOYEES, and PROJECTS. However, only two of these attributes—namely, DNAME and DNUMBER—have basic values; the other four have complex structure and hence build the second level of the complex object structure. One of these four (MGR) has a tuple structure, and the other three (LOCATIONS, EMPLOYEES, PROJECTS) have set structures. At the third level, for a MGR tuple value, we have one

basic attribute (MANAGERSTARTDATE) and one attribute (MANAGER) that refers to an employee object, which has a tuple structure. For a LOCATIONS set, we have a set of basic values, but for both the EMPLOYEES and the PROJECTS sets, we have sets of tuple-structured objects.

Two types of reference semantics exist between a complex object and its components at each level. The first type, which we can call **ownership semantics,** applies when the sub-objects of a complex object are encapsulated within the complex object and are hence considered part of the complex object. The second type, which we can call **reference semantics,** applies when the components of the complex object are themselves independent objects but may be referenced from the complex object. For example, we may consider the DNAME, DNUMBER, MGR, and LOCATIONS attributes to be owned by a DEPARTMENT, whereas EMPLOYEES and PROJECTS are references because they reference independent objects. The first type is also referred to as the *is-part-of* or *is-component-of* relationship; and the second type is called the *is-associated-with* relationship, since it describes an equal association between two independent objects. The is-part-of relationship (ownership semantics) for constructing complex objects has the property that the component objects are encapsulated within the complex object and are considered part of the internal object state. They need not have object identifiers and can only be accessed by methods of that object. They are deleted if the object itself is deleted. On the other hand, referenced components are considered as independent objects that can have their own identity and methods. When a complex object needs to access its referenced components, it must do so by invoking the appropriate methods of the components, since they are not encapsulated within the complex object. Hence, reference semantics represents *relationships* among independent objects. In addition, a referenced component object may be referenced by more than one complex object and hence is not automatically deleted when the complex object is deleted.

An OODBMS should provide storage options for **clustering** the component objects of a complex object together on secondary storage in order to increase the efficiency of operations that access the complex object. In many cases, the object structure is stored on disk pages in an uninterpreted fashion. When a disk page that includes an object is retrieved into memory, the OODBMS can build up the structured complex object from the information on the disk pages, which may refer to additional disk pages that must be retrieved. This is known as **complex object assembly.**

20.6 Other Objected-Oriented Concepts

In this section we give an overview of some additional OO concepts, including polymorphism (operator overloading), multiple inheritance, selective inheritance, versioning, and configurations.

20.6.1 Polymorphism (Operator Overloading)

Another characteristic of OO systems is that they provide for **polymorphism** of operations, which is also known as **operator overloading.** This concept allows the same *operator name* or *symbol* to be bound to two or more different *implementations* of the operator,

depending on the type of objects to which the operator is applied. A simple example from programming languages can illustrate this concept. In some languages, the operator symbol "+" can mean different things when applied to operands (objects) of different types. If the operands of "+" are of type *integer*, the operation invoked is integer addition. If the operands of "+" are of type *floating point*, the operation invoked is floating point addition. If the operands of "+" are of type *set*, the operation invoked is set union. The compiler can determine which operation to execute based on the types of operands supplied.

In OO databases, a similar situation may occur. We can use the GEOMETRY_OBJECT example discussed in Section 20.4 to illustrate polymorphism[26] in OO databases. Suppose that we declare GEOMETRY_OBJECT and its subtypes as follows:

```
GEOMETRY_OBJECT: Shape, Area, ReferencePoint
RECTANGLE subtype-of GEOMETRY_OBJECT (Shape='rectangle'): Width,
Height
TRIANGLE subtype-of GEOMETRY_OBJECT (Shape='triangle'): Side1, Side2,
Angle
CIRCLE subtype-of GEOMETRY_OBJECT (Shape='circle'): Radius
```

Here, the function Area is declared for all objects of type GEOMETRY_OBJECT. However, the implementation of the method for Area may differ for each subtype of GEOMETRY_OBJECT. One possibility is to have a general implementation for calculating the area of a generalized GEOMETRY_OBJECT (for example, by writing a general algorithm to calculate the area of a polygon) and then to rewrite more efficient algorithms to calculate the areas of specific types of geometric objects, such as a circle, a rectangle, a triangle, and so on. In this case, the Area function is *overloaded* by different implementations.

The OODBMS must now select the appropriate method for the Area function based on the type of geometric object to which it is applied. In strongly typed systems, this can be done at compile time, since the object types must be known. This is termed **early** (or **static**) **binding.** However, in systems with weak typing or no typing (such as SMALLTALK and LISP), the type of the object to which a function is applied may not be known until runtime. In this case, the function must check the type of object at runtime and then invoke the appropriate method. This is often referred to as **late** (or **dynamic**) **binding.**

20.6.2 Multiple Inheritance and Selective Inheritance

Multiple inheritance in a type hierarchy occurs when a certain subtype T is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes. For example, we may create a subtype ENGINEERING_MANAGER that is a subtype of both MANAGER and ENGINEER. This leads to the creation of a **type lattice** rather than a type hierarchy. One problem that can occur with multiple inheritance is that the supertypes from which the subtype inherits may have distinct functions of the same name, creating

26. In programming languages, there are several kinds of polymorphism. The interested reader is referred to the bibliographic notes for works that include a more thorough discussion.

an ambiguity. For example, both MANAGER and ENGINEER may have a function called Salary. If the Salary function is implemented by different methods in the MANAGER and ENGINEER supertypes, an ambiguity exists as to which of the two is inherited by the subtype ENGINEERING_MANAGER. It is possible, however, that both ENGINEER and MANAGER inherit Salary from the same supertype (such as EMPLOYEE) higher up in the lattice. The general rule is that if a function is inherited from some *common supertype*, then it is inherited only once. In such a case, there is no ambiguity; the problem only arises if the functions are distinct in the two supertypes.

There are several techniques for dealing with ambiguity in multiple inheritance. One solution is to have the system check for ambiguity when the subtype is created, and to let the user explicitly choose which function is to be inherited at this time. Another solution is to use some system default. A third solution is to disallow multiple inheritance altogether if name ambiguity occurs, instead forcing the user to change the name of one of the functions in one of the supertypes. Indeed, some OO systems do not permit multiple inheritance at all.

Selective inheritance occurs when a subtype inherits only some of the functions of a supertype. Other functions are not inherited. In this case, an EXCEPT clause may be used to list the functions in a supertype that are *not* to be inherited by the subtype. The mechanism of selective inheritance is not typically provided in OO database systems, but it is used more frequently in artificial intelligence applications.[27]

20.6.3 Versions and Configurations

Many database applications that use OO systems require the existence of several **versions** of the same object.[28] For example, consider a database application for a software engineering environment that stores various software artifacts, such as *design modules*, *source code modules*, and *configuration information* to describe which modules should be linked together to form a complex program, and *test cases* for testing the system. Commonly, *maintenance activities* are applied to a software system as its requirements evolve. Maintenance usually involves changing some of the design and implementation modules. If the system is already operational, and if one or more of the modules must be changed, the designer should create a **new version** of each of these modules to implement the changes. Similarly, new versions of the test cases may have to be generated to test the new versions of the modules. However, the existing versions should not be discarded until the new versions have been thoroughly tested and approved; only then should the new versions replace the older ones.

Notice that there may be more than two versions of an object. For example, consider two programmers working to update the same software module concurrently. In this case, two versions, in addition to the original module, are needed. The programmers can update their own versions of the same software module concurrently. This is often

27. In the ODMG model, type inheritance refers to inheritance of operations only, not attributes (see Chapter 21).

28. Versioning is not a problem that is unique to OODBs and it can be applied to relational or other types of DBMSs.

referred to as **concurrent engineering.** However, it eventually becomes necessary to merge these two versions together so that the new (hybrid) version can include the changes made by both programmers. During merging, it is also necessary to make sure that their changes are compatible. This necessitates creating yet another version of the object: one that is the result of merging the two independently updated versions.

As can be seen from the preceding discussion, an OODBMS should be able to store and manage multiple versions of the same conceptual object. Several systems do provide this capability, by allowing the application to maintain multiple versions of an object and to refer explicitly to particular versions as needed. However, the problem of merging and reconciling changes made to two different versions is typically left to the application developers, who know the semantics of the application. Some DBMSs have certain facilities that can compare the two versions with the original object and determine whether any changes made are incompatible, in order to assist with the merging process. Other systems maintain a **version graph** that shows the relationships among versions. Whenever a version v_1 originates by copying another version v, a directed arc can be drawn from v to v_1. Similarly, if two versions v_2 and v_3 are merged to create a new version v_4, directed arcs are drawn from v_2 and v_3 to v_4. The version graph can help users understand the relationships among the various versions and can be used internally by the system to manage the creation and deletion of versions.

When versioning is applied to complex objects, further issues arise that must be resolved. A complex object, such as a software system, may consist of many modules. When versioning is allowed, each of these modules may have a number of different versions and a version graph. A **configuration** of the complex object is a collection consisting of one version of each module arranged in such a way that the module versions in the configuration are compatible and together form a valid version of the complex object. A new version or configuration of the complex object does not have to include new versions for every module. Hence, certain module versions that have not been changed may belong to more than one configuration of the complex object. Notice that a configuration is a collection of versions of *different* objects that together make up a complex object, whereas the version graph describes versions of the *same* object. A configuration should follow the type structure of a complex object; multiple configurations of the same complex object are analogous to multiple versions of a component object.

20.7 SUMMARY

In this chapter we discussed the concepts of the object-oriented approach to database systems, which was proposed to meet the needs of complex database applications and to add database functionality to object-oriented programming languages such as C++. We first discussed the main concepts used in OO databases, which include the following:

- *Object identity:* Objects have unique identities that are independent of their attribute values.
- *Type constructors:* Complex object structures can be constructed by recursively applying a set of basic constructors, such as tuple, set, list, and bag.

- *Encapsulation of operations:* Both the object structure and the operations that can be applied to objects are included in the object class definitions.
- *Programming language compatibility:* Both persistent and transient objects are handled seamlessly. Objects are made persistent by being attached to a persistent collection or by explicit naming.
- *Type hierarchies and inheritance:* Object types can be specified by using a type hierarchy, which allows the inheritance of both attributes and methods of previously defined types. Multiple inheritance is allowed in some models.
- *Extents:* All persistent objects of a particular type can be stored in an extent. Extents corresponding to a type hierarchy have set/subset constraints enforced on them.
- *Support for complex objects:* Both structured and unstructured complex objects can be stored and manipulated.
- *Polymorphism and operator overloading:* Operations and method names can be overloaded to apply to different object types with different implementations.
- *Versioning:* Some OO systems provide support for maintaining several versions of the same object.

In the next chapter, we show how some of these concepts are realized in the ODMG standard.

Review Questions

20.1. What are the origins of the object-oriented approach?
20.2. What primary characteristics should an OID possess?
20.3. Discuss the various type constructors. How are they used to create complex object structures?
20.4. Discuss the concept of encapsulation, and tell how it is used to create abstract data types.
20.5. Explain what the following terms mean in object-oriented database terminology: *method, signature, message, collection, extent.*
20.6. What is the relationship between a type and its subtype in a type hierarchy? What is the constraint that is enforced on extents corresponding to types in the type hierarchy?
20.7. What is the difference between persistent and transient objects? How is persistence handled in typical OO database systems?
20.8. How do regular inheritance, multiple inheritance, and selective inheritance differ?
20.9. Discuss the concept of polymorphism/operator overloading.
20.10. What is the difference between structured and unstructured complex objects?
20.11. What is the difference between ownership semantics and reference semantics in structured complex objects?

20.12. What is versioning? Why is it important? What is the difference between versions and configurations?

Exercises

20.13. Convert the example of GEOMETRY_OBJECTS given in Section 20.4.1 from the functional notation to the notation given in Figure 20.3 that distinguishes between attributes and operations. Use the keyword INHERIT to show that one class inherits from another class.

20.14. Compare inheritance in the EER model (see Chapter 4) to inheritance in the OO model described in Section 20.4.

20.15. Consider the UNIVERSITY EER schema of Figure 4.10. Think of what operations are needed for the entity types/classes in the schema. Do not consider constructor and destructor operations.

20.16. Consider the COMPANY ER schema of Figure 3.2. Think of what operations are needed for the entity types/classes in the schema. Do not consider constructor and destructor operations.

Selected Bibliography

Object-oriented database concepts are an amalgam of concepts from OO programming languages and from database systems and conceptual data models. A number of textbooks describe OO programming languages—for example, Stroustrup (1986) and Pohl (1991) for C++, and Goldberg (1989) for SMALLTALK. Recent books by Cattell (1994) and Lausen and Vossen (1997) describe OO database concepts.

There is a vast bibliography on OO databases, so we can only provide a representative sample here. The October 1991 issue of CACM and the December 1990 issue of *IEEE Computer* describe object-oriented database concepts and systems. Dittrich (1986) and Zaniolo et al. (1986) survey the basic concepts of object-oriented data models. An early paper on object-oriented databases is Baroody and DeWitt (1981). Su et al. (1988) presents an object-oriented data model that is being used in CAD/CAM applications. Mitschang (1989) extends the relational algebra to cover complex objects. Query languages and graphical user interfaces for OO are described in Gyssens et al. (1990), Kim (1989), Alashqur et al. (1989), Bertino et al. (1992), Agrawal et al. (1990), and Cruz (1992).

Polymorphism in databases and object-oriented programming languages is discussed in Osborn (1989), Atkinson and Buneman (1987), and Danforth and Tomlinson (1988). Object identity is discussed in Abiteboul and Kanellakis (1989). OO programming languages for databases are discussed in Kent (1991). Object constraints are discussed in Delcambre et al. (1991) and Elmasri et al. (1993). Authorization and security in OO databases are examined in Rabitti et al. (1991) and Bertino (1992).

Additional references will be given at the end of Chapter 21.

21

Object Database Standards, Languages, and Design

As we discussed at the beginning of Chapter 8, having a standard for a particular type of database system is very important, because it provides support for portability of database applications. **Portability** is generally defined as the capability to execute a particular application program on different systems with minimal modifications to the program itself. In the object database field,[1] portability would allow a program written to access one Object Database Management System (ODBMS) package to access another ODBMS package as long as both packages support the standard faithfully. This is important to database users because they are generally wary of investing in a new technology if the different vendors do not adhere to a standard. To illustrate why portability is important, suppose that a particular user invests thousands of dollars in creating an application that runs on a particular vendor's product and is then dissatisfied with that product for some reason—say the performance does not meet their requirements. If the application was written using the standard language constructs, it is possible for the user to convert the application to a different vendor's product—which adheres to the same language standards but may have better performance for that user's application—without having to do major modifications that require time and a major monetary investment.

1. In this chapter, we will use *object* database instead of *object-oriented* database (as in the previous chapter), since this is now more commonly accepted terminology.

A second potential advantage of having and adhering to standards is that it helps in achieving **interoperability,** which generally refers to the ability of an application to access multiple distinct systems. In database terms, this means that the same application program may access some data stored under one ODBMS package, and other data stored under another package. There are different levels of interoperability. For example, the DBMSs could be two distinct DBMS packages of the same type—for example, two object database systems—or they could be two DBMS packages of different types—say one relational DBMS and one object DBMS. A third advantage of standards is that it allows customers to *compare commercial products* more easily by determining which parts of the standard are supported by each product.

As we discussed in the introduction to Chapter 8, one of the reasons for the success of commercial relational DBMSs is the SQL standard. The lack of a standard for ODBMSs for several years may have caused some potential users to shy away from converting to this new technology. Subsequently, a consortium of ODBMS vendors, called ODMG (Object Data Management Group), proposed a standard that is known as the ODMG-93 or ODMG 1.0 standard. This was revised into ODMG 2.0, which we will describe in this chapter. The standard is made up of several parts: the **object model,** the **object definition language** (*ODL*), the **object query language** (*OQL*), and the **bindings** to object-oriented programming languages. Language bindings have been specified for several object-oriented programming languages including C++, SMALLTALK, and JAVA. Some vendors only offer specific language bindings, without offering the full capabilities of ODL and OQL. We will describe the ODMG object model in Section 21.1, ODL in Section 21.2, OQL in Section 21.3, and the C++ language binding in Section 21.4. Examples of how to use ODL, OQL, and the C++ language binding will use the UNIVERSITY database example introduced in Chapter 4. In our description, we will follow the ODMG 2.0 object model as described in Cattell et al. (1997).[2] It is important to note that many of the ideas embodied in the ODMG object model are based on two decades of research into conceptual modeling and object-oriented databases by many researchers.

Following the description of the ODMG model, we will describe a technique for object database conceptual design in Section 21.5. We will discuss how object-oriented databases differ from relational databases and show how to map a conceptual database design in the EER model to the ODL statements of the ODMG model.

The reader may skip Sections 21.3 through 21.7 if a less detailed introduction to the topic is desired.

21.1 Overview of the Object Model of ODMG

The **ODMG object model** is the data model upon which the object definition language (ODL) and object query language (OQL) are based. In fact, this object model provides the data types, type constructors, and other concepts that can be utilized in the ODL to specify object database schemas. Hence, it is meant to provide a standard data model for object-oriented databases, just as SQL describes a standard data model for relational databases. It

2. The earlier version of the object model was published in 1993.

also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts. We will try to adhere to the ODMG terminology in this chapter. Many of the concepts in the ODMG model have already been discussed in Chapter 20, and we assume the reader has already gone through Sections 20.1 through 20.5. We will point out whenever the ODMG terminology differs from that used in Chapter 20.

21.1.1 Objects and Literals

Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an object identifier and a **state** (or current value), whereas a literal has only a value but *no object identifier*.[3] In either case, the value can have a complex structure. The object state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, that does not change.

An **object** is described by four characteristics: (1) identifier, (2) name, (3) lifetime, and (4) structure. The **object identifier** is a unique system-wide identifier (or OBJECT_ID).[4] Every object must have an object identifier. In addition to the OBJECT_ID, some objects may optionally be given a unique **name** within a particular database—this name can be used to refer to the object in a program, and the system should be able to locate the object given that name.[5] Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object type—such as extents—will have a name. These names are used as **entry points** to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also have unique names. All such names within a particular database must be unique. The **lifetime** of an object specifies whether it is a *persistent object* (that is, a database object) or *transient object* (that is, an object in an executing program that disappears after the program terminates). Finally, the **structure** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is *atomic* or a *collection object*.[6] The term *atomic object* is different than the way we defined the *atom constructor* in Section 20.2.2, and it is quite different from an atomic literal (see below). In the ODMG model, an atomic object is any object that is not a collection, so this also covers *structured objects* created using the *struct* constructor.[7] We will discuss collection objects in Section 21.1.2 and atomic objects in Section 21.1.3. First, we define the concept of a literal.

In the object model, a **literal** is a value that *does not have* an object identifier. However, the value may have a simple or complex structure. There are three types of literals: (1)

3. We will use the terms *value* and *state* interchangeably here.

4. Corresponds to the OID of Chapter 20.

5. This corresponds to the naming mechanism described in Section 20.3.

6. In the ODMG model, *atomic objects* do not correspond to objects whose values are basic data types. All basic values (integers, reals, etc.) are considered to be *literals*.

7. The *struct* construct corresponds to the *tuple constructor* of Chapter 20.

atomic, (2) collection, and (3) structured. **Atomic literals**[8] correspond to the values of basic data types and are predefined. The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords Long, Short, Unsigned Long, Unsigned Short in ODL), regular and double precision floating point numbers (Float, Double), boolean values (Boolean), single characters (Char), character strings (String), and enumeration types (Enum), among others. **Structured literals** correspond roughly to values that are constructed using the tuple constructor described in Section 20.2.2. They include Date, Interval, Time, and Timestamp as built-in structures (see Figure 21.1b), as well as any additional user-defined type structures as needed by each application.[9] User-defined structures are created using the **Struct** keyword in ODL, as in the C and C++ programming languages. **Collection literals** specify a value that is a collection of objects or values but the collection itself does not have an OBJECT_ID. The collections in the object model are SET<T>, BAG<T>, LIST<T>, and ARRAY<T>, where t is the type of objects or values in the collection.[10] Another collection type is Dictionary <K,V>, which is a collection of associations <K,V> where each K is a key (a unique search value) associated with a value V; this can be used to create an index on a collection of values.

Figure 21.1 gives a simplified view of the basic components of the object model. The notation of ODMG uses the keyword *interface* where we had used the keywords *type* and *class* in Chapter 20. In fact, interface is a more appropriate term, since it describes the interface of types of objects—namely, their visible attributes, relationships, and operations.[11] These interfaces are typically noninstantiable (that is, no objects are created for an interface) but they serve to define operations that can be *inherited* by the user-defined objects for a particular application. The keyword *class* in the object model is reserved for user-specified class declarations that form a database schema and are used for creating application objects. Figure 21.1 is a simplified version of the object model. For the full specifications, see Cattell et al. (1997). We will describe the constructs shown in Figure 21.1 as we describe the object model.

```
interface Object {
  ...
  boolean     same_as(in Object other_object);
  Object      copy();
  void        delete();
};
```

FIGURE 21.1A *Overview of the interface definitions for part of the* ODMG *object model. The basic* `Object` *interface, inherited by all objects.*

8. The use of the word atomic in *atomic literal* does correspond to the way we used *atom constructor* in Section 20.2.2.

9. The structures for Date, Interval, Time, and Timestamp can be used to create either literal values or objects with identifiers.

10. These are similar to the corresponding type constructors described in Section 20.2.2.

11. Interface is also the keyword used in the CORBA standard (see Section 21.5) and the JAVA programming language.

```
interface Date : Object {
    enum            Weekday
    {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
    enum            Month
    {January, February, March, April, May, June, July, August, September, October, November, December};
    unsigned short  year();
    unsigned short  month();
    unsigned short  day();
    ...
    boolean         is_equal(in Date other_Date);
    boolean         is_greater(in Date other_Date);
    ...
};

interface Time : Object {
    ...
    unsigned short  hour();
    unsigned short  minute();
    unsigned short  second();
    unsigned short  millisecond();
    ...
    boolean         is_equal(in Time other_Time);
    boolean         is_greater(in Time other_Time);
    ...
    Time            add_interval(in Interval some_Interval);
    Time            subtract_interval(in Interval some_Interval);
    Interval        subtract_time(in Time other_Time);
};

interface Timestamp : Object {
    ...
    unsigned short  year();
    unsigned short  month();
    unsigned short  day();
    unsigned short  hour();
    unsigned short  minute();
    unsigned short  second();
    unsigned short  millisecond();
    ...
    Timestamp       plus(in Interval some_Interval);
    Timestamp       minus(in Interval some_Interval);
    boolean         is_equal(in Timestamp other_Timestamp);
    boolean         is_greater(in Timestamp other_Timestamp);
    ...
};

interface Interval : Object {
    unsigned short  day();
    unsigned short  hour();
    unsigned short  minute();
    unsigned short  second();
    unsigned short  millisecond();
    ...
    Interval        plus(in Interval some_Interval);
    Interval        minus(in Interval some_Interval);
    Interval        product(in long some_value);
    Interval        quotient(in long some_value);
    boolean         is_equal(in Interval other_Interval);
    boolean         is_greater(in Interval other_Interval);
    ...
};
```

FIGURE 21.1B Overview of the interface definitions for part of the ODMG object model. Some standard interfaces for structured literals.

```
interface Collection : Object {
  ...
  exception      ElementNotFound{any element; };
  unsigned long  cardinality();
  boolean        is_empty();
  ...
  boolean        contains_element(in any element);
  void           insert_element(in any element);
  void           remove_element(in any element)
                       raises(ElementNotFound);
  Iterator       create_iterator(in boolean stable);
  ...
};

interface Iterator {
  exception      NoMoreElements();
  ...
  boolean        is_stable();
  boolean        at_end();
  void           reset();
  any            get_element() raises(NoMoreElements);
  void           next_position() raises(NoMoreElements);
  ...
};

interface Set : Collection {
  Set            create_union(in Set other_set);
  ...
  boolean        is_subset_of(in Set other_set);
  ...
};

interface Bag : Collection {
  unsigned long  occurrences_of(in any element);
  Bag            create_union(in Bag other_bag);
  ...
};

interface List : Collection {
  exception      Invalid_Index{unsigned_long index; };
  any            remove_element_at(in unsigned long position)
                       raises(InvalidIndex);
  any            retrieve_element_at(in unsigned long position)
                       raises(InvalidIndex);
  void           replace_element_at(in any element, in unsigned long position)
                       raises(InvalidIndex);
  void           insert_element_after(in any element, in unsigned long position)
                       raises(InvalidIndex);
  ...
  void           insert_element_first(in any element);
  ...
  any            remove_first_element() raises(InvalidIndex);
  ...
  any            retrieve_first_element() raises(InvalidIndex);
  ...
  List           concat(in List other_list);
  void           append(in List other_list);
};
```

FIGURE 21.1C Overview of the interface definitions for part of the ODMG object model. Interface definitions for collection objects.

```
interface Array : Collection {
  exception     Invalid_Index{unsigned_long index; };
  any           remove_element_at(in unsigned long index)
                    raises(InvalidIndex);
  any           retrieve_element_at(in unsigned long index)
                    raises(InvalidIndex);
  void          replace_element_at(in unsigned long index, in any element)
                    raises(InvalidIndex);
  void          resize(in unsigned long new_size);
};

struct Association {any key; any value; };

interface Dictionary : Collection {
  exception     KeyNotFound{any key; };
  void          bind(in any key, in any value);
  void          unbind(in any key) raises(KeyNotFound);
  any           lookup(in any key) raises(KeyNotFound);
  boolean       contains_key(in any key);
};
```

FIGURE 21.1c (CONTINUED)

In the object model, all objects inherit the basic interface of Object, shown in Figure 21.1a. Hence, the basic operations that are inherited by all objects (from the Object interface) are `copy` (creates a new copy of the object), `delete` (deletes the object), and `same_as` (compares the object's identity to another object).[12] In general, operations are applied to objects using the **dot notation.** For example, given an object `o`, to compare it with another object `p`, we write

```
o.same_as(p)
```

The result returned by this expression is Boolean and would be true if the identity of `p` is the same as that of `o`, and false otherwise. Similarly, to create a copy `p` of object `o`, we write

```
p = o.copy()
```

An alternative to the dot notation is the **arrow notation:** `o->same_as(p)` or `o->copy()`.

Type inheritance, which is used to define type/subtype relationships, is specified in the object model using the colon (:) notation, as in the C++ programming language. Hence, in Figure 21.1, we can see that all interfaces, such as Collection, Date, and Time, inherit the basic Object interface. In the object model, there are two main types of objects: (1) collection objects, described in Section 21.1.2, and (2) atomic (and structured) objects, described in Section 21.1.3.

12. Additional operations are defined on objects for *locking* purposes, which are not shown in Figure 21.1. We discuss locking concepts for databases in Chapter 20.

21.1.2 Built-in Interfaces for Collection Objects

Any **collection object** inherits the basic Collection interface shown in Figure 21.1c, which shows the operations for all collection objects. Given a collection object `o`, the `o.cardinality()` operation returns the number of elements in the collection. The operation `o.is_empty()` returns true if the collection `o` is empty, and false otherwise. The operations `o.insert_element(e)` and `o.remove_element(e)` insert or remove an element `e` from the collection `o`. Finally, the operation `o.contains_element(e)` returns true if the collection `o` includes element `e`, and returns false otherwise. The operation `i = o.create_iterator()` creates an **iterator object** `i` for the collection object `o`, which can iterate over each element in the collection. The interface for iterator objects is also shown in Figure 21.1c. The `i.reset()` operation sets the iterator at the first element in a collection (for an unordered collection, this would be some arbitrary element), and `i.next_position()` sets the iterator to the next element. The `i.get_element()` retrieves the **current element,** which is the element at which the iterator is currently positioned.

The ODMG object model uses **exceptions** for reporting errors or particular conditions. For example, the `ElementNotFound` exception in the `Collection` interface would be raised by the `o.remove_element(e)` operation if `e` is not an element in the collection `o`. The `NoMoreElements` exception in the iterator interface would be raised by the `i.next_position()` operation if the iterator is currently positioned at the last element in the collection, and hence no more elements exist for the iterator to point to.

Collection objects are further specialized into `Set`, `List`, `Bag`, `Array`, and `Dictionary`, which inherit the operations of the `Collection` interface. A `Set<t>` object type can be used to create objects such that the value of object `o` is a *set whose elements are of type* `t`. The `Set` interface includes the additional operation `p = o.create_union(s)` (see Figure 21.1c), which returns a new object `p` of type `Set<t>` that is the union of the two sets `o` and `s`. Other operations similar to `create_union` (not shown in Figure 21.1c) are `create_intersection(s)` and `create_difference(s)`. Operations for set comparison include the `o.is_subset_of(s)` operation, which returns true if the set object `o` is a subset of some other set object `s`, and returns false otherwise. Similar operations (not shown in Figure 21.1c) are `is_proper_subset_of(s)`, `is_superset_of(s)`, and `is_proper_superset_of(s)`. The `Bag<t>` object type allows duplicate elements in the collection and also inherits the `Collection` interface. It has three operations—`create_union(b)`, `create_intersection(b)`, and `create_difference(b)`—that all return a new object of type `Bag<t>`. For example, `p = o.create_union(b)` returns a Bag object `p` that is the union of `o` and `b` (keeping duplicates). The `o.occurrences_of(e)` operation returns the number of duplicate occurrences of element `e` in bag `o`.

A `List<t>` object type inherits the `Collection` operations and can be used to create collections where the order of the elements is important. The value of each such object `o` is an *ordered list whose elements are of type* `t`. Hence, we can refer to the first, last, and i^{th} element in the list. Also, when we add an element to the list, we must specify the position in the list where the element is inserted. Some of the `List` operations are shown in Figure 21.1c. If `o` is an object of type `List<t>`, the operation

o.insert_element_first(e) (see Figure 21.1c) inserts the element e before the first element in the list o, so that e becomes the first element in the list. A similar operation (not shown) is o.insert_element_last(e). The operation o.insert_element_after(e,i) in Figure 21.1c inserts the element e after the i^{th} element in the list o and will raise the exception InvalidIndex if no i^{th} element exists in o. A similar operation (not shown) is o.insert_element_before(e,i). To remove elements from the list, the operations are e = o.remove_first_element(), e = o.remove_last_element(), and e = o.remove_element_at(i); these operations remove the indicated element from the list *and* return the element as the operation's result. Other operations retrieve an element without removing it from the list. These are e = o.retrieve_first_element(), e = o.retrieve_last_element(), and e = o.retrieve_element_at(i). Finally, two operations to manipulate lists are defined. These are p = o.concat(l), which creates a new list p that is the concatenation of lists o and l (the elements in list o followed by those in list l), and o.append(l), which appends the elements of list l to the end of list o (without creating a new list object).

The Array<t> object type also inherits the Collection operations. It is similar to a list except that an array has a fixed number of elements. The specific operations for an Array object o are o.replace_element_at(i,e), which replaces the array element at position i with element e; e = o.remove_element_at(i), which retrieves the i^{th} element and replaces it with a null value; and e = o.retrieve_element_at(i), which simply retrieves the i^{th} element of the array. Any of these operations can raise the exception InvalidIndex if i is greater than the array's size. The operation o.resize(n) changes the number of array elements to n.

The last type of collection objects are of type Dictionary<k, v>. This allows the creation of a collection of association pairs <k, v>, where all k (key) values are unique. This allows for associative retrieval of a particular pair given its key value (similar to an index). If o is a collection object of type Dictionary<k,v>, then o.bind(k,v) binds value v to the key k as an association <k,v> in the collection, whereas o.unbind(k) removes the association with key k from o, and v = o.lookup(k) returns the value v associated with key k in o. The latter two operations can raise the exception KeyNotFound. Finally, o.contains_key(k) returns true if key k exists in o, and returns false otherwise.

Figure 21.2 is a diagram that illustrates the inheritance hierarchy of the built-in constructs of the object model. Operations are inherited from the supertype to the subtype. The collection object interfaces described above are *not directly instantiable*; that is, one cannot directly create objects based on these interfaces. Rather, the interfaces can be used to specify user-defined collection objects—of type Set, Bag, List, Array, or Dictionary—for a particular database application. When a user designs a database schema, they will declare their own object interfaces and classes that are relevant to the database application. If an interface or class is one of the collection objects, say a Set, then it will inherit the operations of the Set interface. For example, in a UNIVERSITY database application, the user can specify a class for Set<Student>, whose objects would be sets of Student objects. The programmer can then use the operations for Set<t> to manipulate an object of type Set<Student>. Creating application classes is typically done by utilizing the object definition language ODL (see Section 21.2).

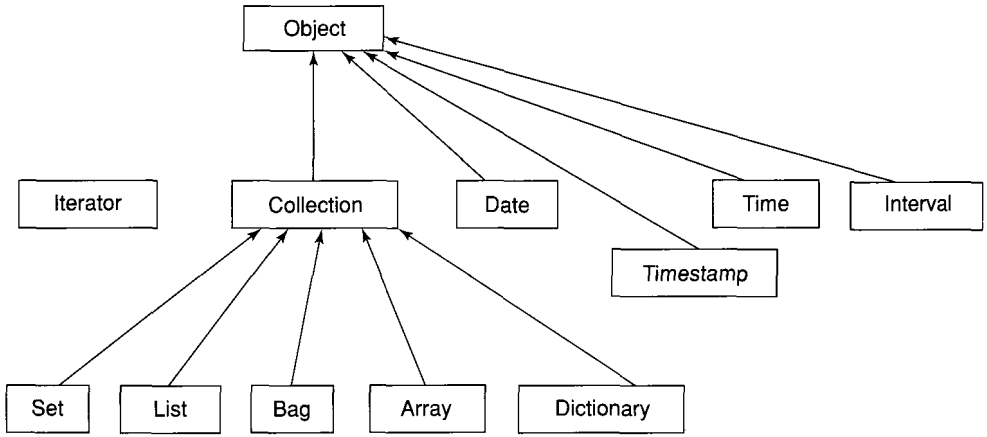


FIGURE 21.2 Inheritance hierarchy for the built-in interfaces of the object model.

It is important to note that all objects in a particular collection *must be of the same type*. Hence, although the keyword **any** appears in the specifications of collection interfaces in Figure 21.1c, this does not mean that objects of any type can be intermixed within the same collection. Rather, it means that any type can be used when specifying the type of elements for a particular collection (including other collection types!).

21.1.3 Atomic (User-Defined) Objects

The previous section described the built-in collection types of the object model. We now discuss how object types for *atomic objects* can be constructed. These are specified using the keyword **class** in ODL. In the object model, any user-defined object that is not a collection object is called an **atomic object.**[13] For example, in a UNIVERSITY database application, the user can specify an object type (class) for Student objects. Most such objects will be **structured objects;** for example, a Student object will have a complex structure, with many attributes, relationships, and operations, but it is still considered atomic because it is not a collection. Such a user-defined atomic object type is defined as a class by specifying its **properties** and **operations.** The properties define the state of the object and are further distinguished into **attributes** and **relationships.** In this subsection, we elaborate on the three types of components—attributes, relationships, and operations—that a user-defined object type for atomic (structured) objects can include. We illustrate our discussion with the two classes `Employee` and `Department` shown in Figure 21.3.

13. As mentioned earlier, this definition of *atomic object* in the ODMG object model is different from the definition of atom constructor given in Chapter 20, which is the definition used in much of the object-oriented database literature.

```
class Employee
( extent all_employees
  key    ssn )
{
  attribute    string                  name;
  attribute    string                  ssn;
  attribute    date                    birthdate;
  attribute    enum Gender{M, F}       sex;
  attribute    short                   age;
  relationship Department              works_for
                    inverse Department::has_emps;
  void         reassign_emp(in string new_dname)
                    raises(dname_not_valid);
};

class Department
( extent all_departments
  key    dname, dnumber )
{
  attribute    string                  dname;
  attribute    short                   dnumber;
  attribute    struct Dept_Mgr {Employee manager, date startdate}
                                       mgr;
  attribute    set<string>             locations;
  attribute    struct Projs {string projname, time weekly_hours}
                                       projs;
  relationship set<Employee>           has_emps inverse Employee::works_for;
  void         add_emp(in string new_ename) raises(ename_not_valid);
  void         change_manager(in string new_mgr_name; in date startdate);
};
```

FIGURE 21.3 The attributes, relationships, and operations in a class definition.

An **attribute** is a property that describes some aspect of an object. Attributes have values, which are typically literals having a simple or complex structure, that are stored within the object. However, attribute values can also be `Object_Id`s of other objects. Attribute values can even be specified via methods that are used to calculate the attribute value. In Figure 21.3,[14] the attributes for `Employee` are `name`, `ssn`, `birthdate`, `sex`, and `age`, and those for `Department` are `dname`, `dnumber`, `mgr`, `locations`, and `projs`. The `mgr` and `projs` attributes of `Department` have complex structure and are defined via **struct,** which corresponds to the *tuple constructor* of Chapter 20. Hence, the value of `mgr` in each `Department` object will have two components: `manager`, whose value is an `Object_Id` that references the `Employee` object that manages the `Department`, and `startdate`, whose value is a `date`. The locations attribute of `Department` is defined via the `set` constructor, since each `Department` object can have a set of locations.

14. We are using the Object Definition Language (ODL) notation in Figure 21.3, which will be discussed in more detail in Section 21.2.

A **relationship** is a property that specifies that two objects in the database are related together. In the object model of ODMG, only binary relationships (see Chapter 3) are explicitly represented, and each binary relationship is represented by a *pair of inverse references* specified via the keyword relationship. In Figure 21.3, one relationship exists that relates each `Employee` to the `Department` in which he or she works—the `works_for` relationship of `Employee`. In the inverse direction, each `Department` is related to the set of `Employees` that work in the `Department`—the `has_emps` relationship of `Department`. The keyword `inverse` specifies that these two properties specify a single conceptual relationship in inverse directions.[15] By specifying inverses, the database system can maintain the referential integrity of the relationship automatically. That is, if the value of `works_for` for a particular `Employee e` refers to `Department d`, then the value of `has_emps` for `Department d` must include a reference to `e` in its set of `Employee` references. If the database designer desires to have a relationship to be represented in *only one direction*, then it has to be modeled as an attribute (or operation). An example is the `manager` component of the `mgr` attribute in `Department`.

In addition to attributes and relationships, the designer can include **operations** in object type (class) specifications. Each object type can have a number of **operation signatures,** which specify the operation name, its argument types, and its returned value, if applicable. Operation names are unique within each object type, but they can be overloaded by having the same operation name appear in distinct object types. The operation signature can also specify the names of **exceptions** that can occur during operation execution. The implementation of the operation will include the code to raise these exceptions. In Figure 21.3, the `Employee` class has one operation, `reassign_emp`, and the `Department` class has two operations, `add_emp` and `change_manager`.

21.1.4 Interfaces, Classes, and Inheritance

In the ODMG object model, two concepts exist for specifying object types: interfaces and classes. In addition, two types of inheritance relationships exist. In this section, we discuss the differences and similarities among these concepts. Following the ODMG terminology, we use the word **behavior** to refer to *operations*, and **state** to refer to *properties* (attributes and relationships).

An **interface** is a specification of the abstract behavior of an object type, which specifies the operation signatures. Although an interface may have state properties (attributes and relationships) as part of its specifications, these *cannot* be inherited from the interface, as we shall see. An interface also is **noninstantiable**—that is, one cannot create objects that correspond to an interface definition.[16]

A **class** is a specification of both the abstract behavior and abstract state of an object type, and is **instantiable**—that is, one can create individual object instances corresponding

15. Chapter 3 discussed how a relationship can be represented by two attributes in inverse directions.

16. This is somewhat similar to the concept of abstract class in the C++ programming language.

to a class definition. Because interfaces are noninstantiable, they are mainly used to specify abstract operations that can be inherited by classes or by other interfaces. This is called **behavior inheritance** and is specified by the ":" symbol.[17] Hence, in the ODMG object model, behavior inheritance requires the supertype to be an interface, whereas the subtype could be either a class or another interface.

Another inheritance relationship, called EXTENDS and specified by the **extends** keyword, is used to inherit both state and behavior strictly among classes. In an EXTENDS inheritance, both the supertype and the subtype must be classes. Multiple inheritance via EXTENDS is not permitted. However, multiple inheritance is allowed for behavior inheritance via ":". Hence, an interface may inherit behavior from several other interfaces. A class may also inherit behavior from several interfaces via ":", in addition to inheriting behavior and state from *at most one* other class via EXTENDS. We will give examples in Section 21.2 of how these two inheritance relationships—":" and EXTENDS—may be used.

21.1.5 Extents, Keys, and Factory Objects

In the ODMG object model, the database designer can declare an **extent** for any object type that is defined via a **class** declaration. The extent is given a name, and it will contain all persistent objects of that class. Hence, the extent behaves as a *set object* that holds all persistent objects of the class. In Figure 21.3, the `Employee` and `Department` classes have extents called `all_employees` and `all_departments`, respectively. This is similar to creating two objects—one of type `Set<Employee>` and the second of type `Set<Department>`—and making them persistent by naming them `all_employees` and `all_departments`. Extents are also used to automatically enforce the set/subset relationship between the extents of a supertype and its subtype. If two classes `A` and `B` have extents `all_A` and `all_B`, and class `B` is a subtype of class `A` (that is, class `B` EXTENDS class `A`), then the collection of objects in `all_B` must be a subset of those in `all_A` at any point in time. This constraint is automatically enforced by the database system.

A class with an extent can have one or more keys. A **key** consists of one or more properties (attributes or relationships) whose values are constrained to be unique for each object in the extent. For example, in Figure 21.3, the `Employee` class has the `ssn` attribute as key (each `Employee` object in the extent must have a unique `ssn` value), and the `Department` class has two distinct keys: `dname` and `dnumber` (each `Department` must have a unique `dname` and a unique `dnumber`). For a composite key[18] that is made of several properties, the properties that form the key are contained in parentheses. For example, if a class `Vehicle` with an extent `all_vehicles` has a key made up of a combination of two

17. The ODMG report also calls interface inheritance as type/subtype, is-a, and generalization/specialization relationships, although, in the literature, these terms have been used to describe inheritance of both state and operations (see Chapters 4 and 20).

18. A composite key is called a *compound key* in the ODMG report.

attributes `state` and `license_number`, they would be placed in parentheses as `(state, license_number)` in the key declaration.

Next, we present the concept of **factory object**—an object that can be used to generate or create individual objects via its operations. Some of the interfaces of factory objects that are part of the ODMG object model are shown in Figure 21.4. The interface `ObjectFactory` has a single operation, `new()`, which returns a new object with an `Object_Id`. By inheriting this interface, users can create their own factory interfaces for each user-defined (atomic) object type, and the programmer can implement the operation *new* differently for each type of object. Figure 21.4 also shows a `DateFactory` interface, which has additional operations for creating a new `calendar_date`, and for creating an object whose value is the `current_date`, among other operations (not shown in Figure 21.4). As we can see, a factory object basically provides the **constructor operations** for new objects.

Finally, we discuss the concept of a **database.** Because a ODBMS can create many different databases, each with its own schema, the ODMG object model has interfaces for `DatabaseFactory` and `Database` objects, as shown in Figure 21.4. Each database has its own *database name*, and the **bind** operation can be used to assign individual unique names to persistent objects in a particular database. The **lookup** operation returns an object from the database that has the specified `object_name`, and the **unbind** operation removes the name of a persistent named object from the database.

```
interface ObjectFactory {
  Object          new();
};

interface DateFactory : ObjectFactory {
  exception       InvalidDate{};
  ...
  Date            calendar_date(  in unsigned short year,
                                  in unsigned short month,
                                  in unsigned short day)
                  raises(InvalidDate);
  ...
  Date            current();
};

interface DatabaseFactory {
  Database        new();
};

interface Database {
  void            open(in string database_name);
  void            close();
  void            bind(in any some_object, in string object_name);
  Object          unbind(in string name);
  Object          lookup(in string object_name)
                        raises(ElementNotFound);
  ...
};
```

FIGURE 21.4 Interfaces to illustrate factory objects and database objects.

21.2 THE OBJECT DEFINITION LANGUAGE ODL

After our overview of the ODMG object model in the previous section, we now show how these concepts can be utilized to create an object database schema using the object definition language ODL.[19] The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any particular programming language. Its main use is to create object specifications—that is, classes and interfaces. Hence, ODL is not a full programming language. A user can specify a database schema in ODL independently of any programming language, then use the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming languages, such as C++, SMALLTALK, and JAVA. We will give an overview of the C++ binding in Section 21.4.

Figure 21.5b shows a possible object schema for part of the UNIVERSITY database, which was presented in Chapter 4. We will describe the concepts of ODL using this example, and the one in Figure 21.7. The graphical notation for Figure 21.5b is shown in Figure 21.5a and can be considered as a variation of EER diagrams (see Chapter 4) with the added concept of interface inheritance but without several EER concepts, such as categories (union types) and attributes of relationships.

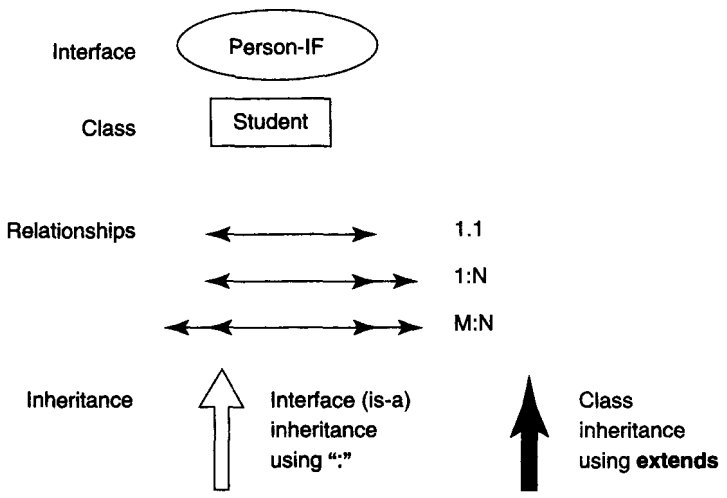
Figure 21.6 shows one possible set of ODL class definitions for the UNIVERSITY database. In general, there may be several possible mappings from an object schema diagram (or EER schema diagram) into ODL classes. We will discuss these options further in Section 21.5.

Figure 21.6 shows the straightforward way of mapping part of the UNIVERSITY database from Chapter 4. Entity types are mapped into ODL classes, and inheritance is done using *EXTENDS*. However, there is no direct way to map categories (union types) or to do multiple inheritance. In Figure 21.6, the classes `Person`, `Faculty`, `Student`, and `GradStudent` have the extents `persons`, `faculty`, `students`, and `grad_students`, respectively. Both `Faculty` and `Student` EXTENDS `Person`, and `GradStudent` EXTENDS `Student`. Hence, the collection of `students` (and the collection of `faculty`) will be constrained to be a subset of the collection of `persons` at any point in time. Similarly, the collection of `grad_students` will be a subset of `students`. At the same time, individual `Student` and `Faculty` objects will inherit the properties (attributes and relationships) and operations of `Person`, and individual `GradStudent` objects will inherit those of `Student`.

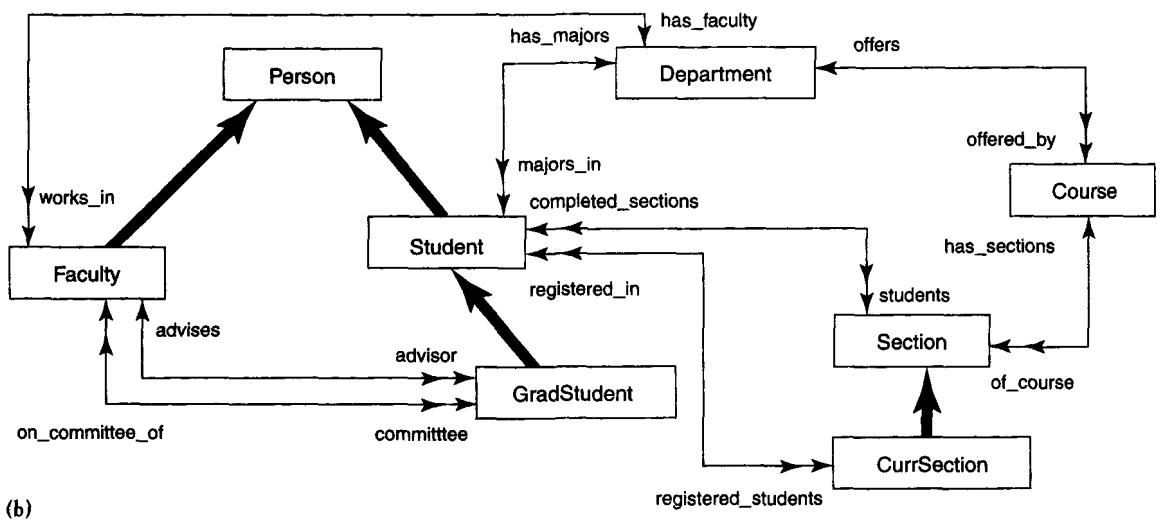
The classes `Department`, `Course`, `Section`, and `CurrSection` in Figure 21.6 are straightforward mappings of the corresponding entity types in Figure 21.5b. However, the class `Grade` requires some explanation. The `Grade` class corresponds to the M:N relationship between `Student` and `Section` in Figure 21.5b. The reason it was made into a separate class (rather than as a pair of inverse relationships) is because it includes the relationship attribute `grade`.[20] Hence, the M:N relationship is mapped to the class `Grade`, and a pair of 1:N relationships, one between `Student` and `Grade` and the other between

19. The ODL syntax and data types are meant to be compatible with the Interface Definition Language (IDL) of CORBA (Common Object Request Broker Architecture), with extensions for relationships and other database concepts.

20. We will discuss alternative mappings for attributes of relationships in Section 21.5.



(a)



(b)

FIGURE 21.5 An example of a database schema. (a) Graphical notation for representing ODL schemas. (b) A graphical object database schema for part of the UNIVERSITY database.

```
class Person
( extent persons
  key   ssn )
{
  attribute    struct Pname {string fname, string mname, string lname }
                                    name;
  attribute    string               ssn;
  attribute    date                 birthdate;
  attribute    enum Gender{M, F}    sex;
  attribute    struct Address
               {short no, string street, short aptno, string city, string state, short zip }
                                    address;
  short   age();
};

class Faculty extends Person
( extent faculty )
{
  attribute    string          rank;
  attribute    float           salary;
  attribute    string          office;
  attribute    string          phone;
  relationship Department      works_in inverse Department::has_faculty;
  relationship set<GradStudent> advises inverse GradStudent::advisor;
  relationship set<GradStudent> on_committee_of
                               inverse GradStudent::committee;
  void    give_raise(in float raise);
  void    promote(in string new_rank);
};

class Grade
( extent grades )
{
  attribute    enum GradeValues{A,B,C,D,F,I,P}
                               grade;
  relationship Section section inverse Section::students;
  relationship Student student inverse Student::completed_sections;
};

class Student extends Person
( extent students )
{
  attribute    string          class;
  attribute    Department      minors_in;
  relationship Department majors_in inverse Department::has_majors;
  relationship set<Grade> completed_sections inverse Grade::student;
  relationship set<CurrSection> registered_in
                               inverse CurrSection::registered_students;
  void    change_major(in string dname) raises(dname_not_valid);
  float   gpa();
  void    register(in short secno) raises(section_not_valid);
  void    assign_grade(in short secno; in GradeValue grade)
                               raises(section_not_valid,grade_not_valid);
```

FIGURE 21.6 Possible ODL schema for the UNIVERSITY database of Figure 21.5(b).

```
class Degree
{
    attribute    string          college;
    attribute    string          degree;
    attribute    string          year;
};

class GradStudent extends Student
(   extent grad_students   )
{
    attribute    set<Degree>     degrees;
    relationship Faculty advisor inverse Faculty::advises;
    relationship set<Faculty> committee inverse Faculty::on_committee_of;
    void    assign_advisor(in string lname; in string fname)
                                 raises(faculty_not_valid);
    void    assign_committee_member(in string lname; in string fname)
                                 raises(faculty_not_valid);
};

class Department
(   extent departments key dname   )
{
    attribute    string          dname;
    attribute    string          dphone;
    attribute    string          doffice;
    attribute    string          college;
    attribute    Faculty         chair;
    relationship set<Faculty> has_faculty inverse Faculty::works_in;
    relationship set<Student> has_majors inverse Student::majors_in;
    relationship set<Course> offers inverse Course::offered_by;
};

class Course
(   extent courses key cno   )
{
    attribute    string          cname;
    attribute    string          cno;
    attribute    string          description;
    relationship set<Section> has_sections inverse Section::of_course;
    relationship Department offered_by inverse Department::offers;
};

class Section
(   extent sections   )
{
    attribute    short           secno;
    attribute    string          year;
    attribute    enum Quarter{Fall, Winter, Spring, Summer} qtr;
    relationship set<Grade> students inverse Grade::section;
    relationship Course of_course inverse Course::has_sections;
};

class CurrSection extends Section
(   extent current_sections      )
{
    relationship set<Student> registered_students inverse Student::registered_in
    void    register_student(in string ssn)
                raises(student_not_valid, section_full);
};
```

FIGURE 21.6 (CONTINUED)

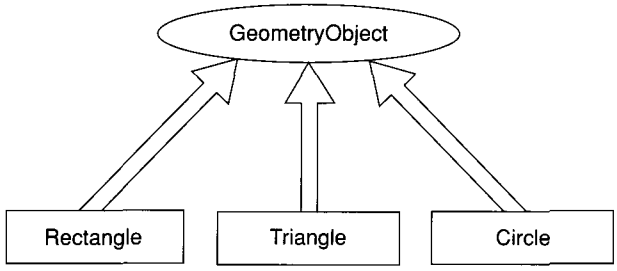


FIGURE 21.7A An illustration of interface inheritance via ":". Graphical schema representation.

`Section` and `Grade`.[21] These two relationships are represented by the following relationship properties: `completed_sections` of `Student`; `section` and `student` of `Grade`; and `students` of `Section` (see Figure 21.6). Finally, the class `Degree` is used to represent the composite, multivalued attribute degrees of `GradStudent` (see Figure 4.10).

Because the previous example did not include any interfaces, only classes, we now utilize a different example to illustrate interfaces and interface (behavior) inheritance. Figure 21.7 is part of a database schema for storing geometric objects. An interface `GeometryObject` is specified, with operations to calculate the `perimeter` and `area` of a geometric object, plus operations to `translate` (move) and `rotate` an object. Several classes (`Rectangle`, `Triangle`, `Circle`, . . .) inherit the `GeometryObject` interface. Since `GeometryObject` is an interface, it is *noninstantiable*—that is, no objects can be created based on this interface directly. However, objects of type `Rectangle`, `Triangle`, `Circle`, . . . can be created, and these objects inherit all the operations of the `GeometryObject` interface. Note that with interface inheritance, only operations are inherited, not properties (attributes, relationships). Hence, if a property is needed in the inheriting class, it must be repeated in the class definition, as with the `reference_point` attribute in Figure 21.7. Notice that the inherited operations can have different implementations in each class. For example, the implementations of the `area` and `perimeter` operations may be different for `Rectangle`, `Triangle`, and `Circle`.

Multiple inheritance of interfaces by a class is allowed, as is multiple inheritance of interfaces by another interface. However, with the EXTENDS (class) inheritance, multiple inheritance is *not permitted*. Hence, a class can inherit via EXTENDS from at most one class (in addition to inheriting from zero or more interfaces).

21. This is similar to how an M:N relationship is mapped in the relational model (see Chapter 7) and in the legacy network model (see Appendix C).

```
interface GeometryObject
{
   attribute     enum Shape{Rectangle,Triangle,Circle,...} shape;
   attribute     struct Point {short x, short y}     reference_point;
   float   perimeter();
   float   area();
   void    translate(in short x_translation; in short y_translation);
   void    rotate(in float angle_of_rotation);
};

class Rectangle : GeometryObject
(  extent rectangles     )
{
   attribute     struct Point {short x, short y}     reference_point;
   attribute     short               length;
   attribute     short               height;
   attribute     float               orientation_angle;
};

class Triangle : GeometryObject
(  extent triangles  )
{
   attribute     struct Point {short x, short y}     reference_point;
   attribute     short               side_1;
   attribute     short               side_2;
   attribute     float               side1_side2_angle;
   attribute     float               side1_orientation_angle;
};

class Circle : GeometryObject
(  extent circles   )
{
   attribute     struct Point {short x, short y}     reference_point;
   attribute     short               radius;
};

...
```

FIGURE 21.7B An illustration of interface inheritance via ":". Corresponding interface and class definitions in ODL.

21.3 THE OBJECT QUERY LANGUAGE OQL

The object query language (OQL) is the query language proposed for the ODMG object model. It is designed to work closely with the programming languages for which an ODMG binding is defined, such as C++, SMALLTALK, and JAVA. Hence, an OQL query embedded into one of these programming languages can return objects that match the type system of that language. In addition, the implementations of class operations in an ODMG schema can have their code written in these programming languages. The OQL syntax for queries is similar to the syntax of the relational standard query language SQL, with additional features for ODMG concepts, such as object identity, complex objects, operations, inheritance, polymorphism, and relationships.

We will first discuss the syntax of simple OQL queries and the concept of using named objects or extents as database entry points in Section 21.3.1. Then in Section 21.3.2, we discuss the structure of query results and the use of path expressions to traverse relationships among objects. Other OQL features for handling object identity, inheritance, polymorphism, and other object oriented concepts are discussed in Section 21.3.3. The examples to illustrate OQL queries are based on the UNIVERSITY database schema given in Figure 21.6.

21.3.1 Simple OQL Queries, Database Entry Points, and Iterator Variables

The basic OQL syntax is a **select** . . . **from** . . . **where** . . . structure, as for SQL. For example, the query to retrieve the names of all departments in the college of 'Engineering' can be written as follows:

```
Q0:  SELECT   d.dname
     FROM     d in departments
     WHERE    d.college = 'Engineering';
```

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object*. For many queries, the entry point is the name of the **extent** of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. Looking at the extent names in Figure 21.6, the named object `departments` is of type `set<Department>`; `persons` is of type `set<Person>`; `faculty` is of type `set<Faculty>`; and so on.

The use of an extent name—`departments` in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable**[22]—`d` in Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the where-clause. In Q0, only persistent objects `d` in the collection of `departments` that satisfy the condition `d.college = 'Engineering'` are selected for the query result. For each selected object `d`, the value of `d.dname` is retrieved in the query result. Hence, the *type of the result* for Q0 is `bag<string>`, because the type of each `dname` value is string (even though the actual result is a set because `dname` is a key attribute). In general, the result of a query would be of type *bag* for `select . . . from . . .` and of type *set* for `select distinct . . . from . . .`, as in SQL (adding the keyword `distinct` eliminates duplicates).

Using the example in Q0, there are three syntactic options for specifying iterator variables:

d **in** departments

departments d

departments **as** d

22. This is similar to the tuple variables that range over tuples in SQL queries.

We will use the first construct in our examples.[23]

The named objects used as database entry points for OQL queries are not limited to the names of extents. Any named persistent object, whether it refers to an atomic (single) object or to a collection object can be used as a database entry point.

21.3.2 Query Results and Path Expressions

The result of a query can in general be of any type that can be expressed in the ODMG object model. A query does not have to follow the `select . . . from . . . where . . .` structure; in the simplest case, any persistent name on its own is a query, whose result is a reference to that persistent object. For example, the query

```
Q1: departments;
```

returns a reference to the collection of all persistent department objects, whose type is `set<Department>`. Similarly, suppose we had given (via the database bind operation, see Figure 21.4) a persistent name `csdepartment` to a single department object (the computer science department); then, the query:

```
Q1a: csdepartment;
```

returns a reference to that individual object of type `Department`. Once an entry point is specified, the concept of a **path expression** can be used to specify a *path* to related attributes and objects. A path expression typically starts at a *persistent object name*, or at the iterator variable that ranges over individual objects in a collection. This name will be followed by zero or more relationship names or attribute names connected using the *dot notation*. For example, referring to the UNIVERSITY database of Figure 21.6, the following are examples of path expressions, which are also valid queries in OQL:

```
Q2:  csdepartment.chair;
Q2a: csdepartment.chair.rank;
Q2b: csdepartment.has_faculty;
```

The first expression Q2 returns an object of type `Faculty`, because that is the type of the attribute `chair` of the `Department` class. This will be a reference to the `Faculty` object that is related to the department object whose persistent name is `csdepartment` via the attribute `chair`; that is, a reference to the `Faculty` object who is chairperson of the computer science department. The second expression Q2a is similar, except that it returns the `rank` of this `Faculty` object (the computer science chair) rather than the object reference; hence, the type returned by Q2a is string, which is the data type for the `rank` attribute of the `Faculty` class.

Path expressions Q2 and Q2a return single values, because the attributes `chair` (of `Department`) and `rank` (of `Faculty`) are both single-valued and they are applied to a single object. The third expression Q2b is different; it returns an object of type `set<Faculty>` even when applied to a single object, because that is the type of the relationship `has_faculty` of the `Department` class. The collection returned will include

23. Note that the latter two options are similar to the syntax for specifying tuple variables in SQL queries.

references to all Faculty objects that are related to the department object whose persistent name is csdepartment via the relationship has_faculty; that is, references to all Faculty objects who are working in the computer science department. Now, to return the ranks of computer science faculty, we *cannot* write

```
Q3': csdepartment.has_faculty.rank;
```

This is because it is not clear whether the object returned would be of type set<string> or bag<string> (the latter being more likely, since multiple faculty may share the same rank). Because of this type of ambiguity problem, OQL does not allow expressions such as Q3'. Rather, one must use an iterator variable over these collections, as in Q3a or Q3b below:

```
Q3a: select  f.rank
     from    f in csdepartment.has_faculty;
Q3b: select  distinct f.rank
     from    f in csdepartment.has_faculty;
```

Here, Q3a returns bag<string> (duplicate rank values appear in the result), whereas Q3b returns set<string> (duplicates are eliminated via the distinct keyword). Both Q3a and Q3b illustrate how an iterator variable can be defined in the from-clause to range over a restricted collection specified in the query. The variable f in Q3a and Q3b ranges over the elements of the collection csdepartment.has_faculty, which is of type set<Faculty>, and includes only those faculty that are members of the computer science department.

In general, an OQL query can return a result with a complex structure specified in the query itself by utilizing the struct keyword. Consider the following two examples:

```
Q4:  csdepartment.chair.advises;
Q4a: select struct (name:struct(last_name: s.name.lname,
                                first_name: s.name.fname),
                    degrees:(select struct (deg: d.degree,
                                            yr: d.year,
                                            college: d.college)
                             from d in s.degrees)
     from s in csdepartment.chair.advises;
```

Here, Q4 is straightforward, returning an object of type set<GradStudent> as its result; this is the collection of graduate students that are advised by the chair of the computer science department. Now, suppose that a query is needed to retrieve the last and first names of these graduate students, plus the list of previous degrees of each. This can be written as in Q4a, where the variable s ranges over the collection of graduate students advised by the chairperson, and the variable d ranges over the degrees of each such student s. The type of the result of Q4a is a collection of (first-level) structs where each struct has two components: name and degrees.[24] The name component is a further struct made up of last_name and first_name, each being a single string. The degrees component is defined by an embedded query and is itself a

24. As mentioned earlier, struct corresponds to the tuple constructor discussed in Chapter 20.

collection of further (second level) structs, each with three string components: deg, yr, and college.

Note that OQL is *orthogonal* with respect to specifying path expressions. That is, attributes, relationships, and operation names (methods) can be used interchangeably within the path expressions, as long as the type system of OQL is not compromised. For example, one can write the following queries to retrieve the grade point average of all senior students majoring in computer science, with the result ordered by gpa, and within that by last and first name:

```
Q5a: select struct (last_name: s.name.lname, first_name:
                    s.name.fname, gpa: s.gpa)
     from s in csdepartment.has_majors
     where s.class = 'senior'
     order by gpa desc, last_name asc, first_name asc;
Q5b: select struct (last_name: s.name.lname, first_name:
                    s.name.fname, gpa: s.gpa)
     from s in students
     where s.majors_in.dname = 'Computer Science' and
           s.class = 'senior'
     order by gpa desc, last_name asc, first_name asc;
```

Q5a used the named entry point csdepartment to directly locate the reference to the computer science department and then locate the students via the relationship has_majors, whereas Q5b searches the students extent to locate all students majoring in that department. Notice how attribute names, relationship names, and operation (method) names are all used interchangeably (in an orthogonal manner) in the path expressions: gpa is an operation; majors_in and has_majors are relationships; and class, name, dname, lname, and fname are attributes. The implementation of the gpa operation computes the grade point average and returns its value as a float type for each selected student.

The order by clause is similar to the corresponding SQL construct, and specifies in which order the query result is to be displayed. Hence, the collection returned by a query with an order by clause is of type *list*.

21.3.3 Other Features of OQL

Specifying Views as Named Queries. The view mechanism in OQL uses the concept of a **named query.** The *define* keyword is used to specify an identifier of the named query, which must be a unique name among all named objects, class names, method names, or function names in the schema. If the identifier has the same name as an existing named query, then the new definition replaces the previous definition. Once defined, a query definition is persistent until it is redefined or deleted. A view can also have parameters (arguments) in its definition.

For example, the following view V1 defines a named query has_minors to retrieve the set of objects for students minoring in a given department:

```
V1: define has_minors(deptname) as
    select s
    from s in students
    where s.minors_in.dname = deptname;
```

Because the ODL schema in Figure 21.6 only provided a unidirectional minors_in attribute for a Student, we can use the above view to represent its inverse without having to explicitly define a relationship. This type of view can be used to represent inverse relationships that are not expected to be used frequently. The user can now utilize the above view to write queries such as

```
has_minors('Computer Science');
```

which would return a bag of students minoring in the Computer Science department. Note that in Figure 21.6, we did define has_majors as an explicit relationship, presumably because it is expected to be used more often.

Extracting Single Elements from Singleton Collections.

An OQL query will, in general, return a collection as its result, such as a bag, set (if distinct is specified), or list (if the order by clause is used). If the user requires that a query only return a single element, there is an *element* operator in OQL that is guaranteed to return a single element e from a singleton collection c that contains only one element. If c contains more than one element or if c is empty, then the element operator *raises an exception*. For example, Q6 returns the single object reference to the computer science department:

```
Q6: element (select d
             from d in departments
             where d.dname = 'Computer Science');
```

Since a department name is unique across all departments, the result should be one department. The type of the result is d:Department.

Collection Operators (Aggregate Functions, Quantifiers).

Because many query expressions specify collections as their result, a number of operators have been defined that are applied to such collections. These include aggregate operators as well as membership and quantification (universal and existential) over a collection.

The aggregate operators (min, max, count, sum, and avg) operate over a collection.[25] The operator count returns an integer type. The remaining aggregate operators (min, max, sum, avg) return the same type as the type of the operand collection. Two examples follow. The query Q7 returns the number of students minoring in 'Computer Science,' while Q8 returns the average gpa of all seniors majoring in computer science.

25. These correspond to aggregate functions in SQL.

```
Q7:  count (s in has_minors('Computer Science'));
Q8:  avg (select s.gpa
          from s in students
          where s.majors_in.dname = 'Computer Science' and
                s.class = 'senior');
```

Notice that aggregate operations can be applied to any collection of the appropriate type and can be used in any part of a query. For example, the query to retrieve all department names that have more that 100 majors can be written as in Q9:

```
Q9:  select d.dname
     from   d in departments
     where  count (d.has_majors) > 100;
```

The *membership* and *quantification* expressions return a boolean type—that is, true or false. Let v be a variable, c a collection expression, b an expression of type boolean (that is, a boolean condition), and e an element of the type of elements in collection c. Then:

(e **in** c) returns true if element e is a member of collection c.

(**for all** v **in** c: b) returns true if *all* the elements of collection c satisfy b.

(**exists** v **in** c: b) returns true if there is at least one element in c satisfying b.

To illustrate the membership condition, suppose we want to retrieve the names of all students who completed the course called 'Database Systems I'. This can be written as in Q10, where the nested query returns the collection of course names that each student s has completed, and the membership condition returns true if 'Database Systems I' is in the collection for a particular student s:

```
Q10: select s.name.lname, s.name.fname
     from s in students
     where 'Database Systems I' in
           (select c.cname
            from c in s.completed_sections.section.of_course);
```

Q10 also illustrates a simpler way to specify the select clause of queries that return a collection of structs; the type returned by Q10 is `bag<struct(string, string)>`.

One can also write queries that return true/false results. As an example, let us assume that there is a named object called `Jeremy` of type `Student`. Then, query Q11 answers the following question: "Is Jeremy a computer science minor?" Similarly, Q12 answers the question "Are all computer science graduate students advised by computer science faculty?". Both Q11 and Q12 return true or false, which are interpreted as yes or no answers to the above questions:

```
Q11: Jeremy in has_minors('Computer Science');
Q12: for all g in
       (select s
        from s in grad_students
        where s.majors_in.dname = 'Computer Science')
     : g.advisor in csdepartment.has_faculty;
```

Note that query Q12 also illustrates how attribute, relationship, and operation inheritance applies to queries. Although s is an iterator that ranges over the extent grad_students, we can write s.majors_in because the majors_in relationship is inherited by GradStudent from Student via EXTENDS (see Figure 21.6). Finally, to illustrate the exists quantifier, query Q13 answers the following question: "Does any graduate computer science major have a 4.0 gpa?" Here, again, the operation gpa is inherited by GradStudent from Student via EXTENDS.

```
Q13: exists g in
       (select s
        from s in grad_students
        where s.majors_in.dname = 'Computer Science')
     : g.gpa = 4;
```

Ordered (Indexed) Collection Expressions. As we discussed in Section 21.1.2, collections that are lists and arrays have additional operations, such as retrieving the i^{th}, first and last elements. In addition, operations exist for extracting a subcollection and concatenating two lists. Hence, query expressions that involve lists or arrays can invoke these operations. We will illustrate a few of these operations using example queries. Q14 retrieves the last name of the faculty member who earns the highest salary:

```
Q14: first (select    struct(faculty: f.name.lname, salary:
                          f.salary)
            from      f in faculty
            order by  f.salary desc);
```

Q14 illustrates the use of the *first* operator on a list collection that contains the salaries of faculty members sorted in descending order on salary. Thus the first element in this sorted list contains the faculty member with the highest salary. This query assumes that only one faculty member earns the maximum salary. The next query, Q15, retrieves the top three computer science majors based on gpa.

```
Q15: (select struct(last_name: s.name.lname, first_name:
                    s.name.fname, gpa: s.gpa)
     from s in csdepartment.has_majors
     order by gpa desc) [0:2];
```

The select-from-order-by query returns a list of computer science students ordered by gpa in descending order. The first element of an ordered collection has an index position of 0, so the expression [0:2] returns a list containing the first, second and third elements of the select-from-order-by result.

The Grouping Operator. The group by clause in OQL, although similar to the corresponding clause in SQL, provides explicit reference to the collection of objects within each *group* or *partition*. First we give an example, then describe the general form of these queries.

Q16 retrieves the number of majors in each department. In this query, the students are grouped into the same partition (group) if they have the same major; that is, the same value for s.majors_in.dname:

```
Q16: select    struct(deptname, number_of_majors:
                      count (partition))
     from      s in students
     group by  deptname: s.majors_in.dname;
```

The result of the grouping specification is of type set<struct(deptname: string, partition: bag<struct(s:Student)>)>, which contains a struct for each group (PARTITION) that has two components: the grouping attribute value (deptname) and the bag of the student objects in the group (partition). The select clause returns the grouping attribute (name of the department), and a count of the number of elements in each partition (that is, the number of students in each department), where **partition** is the keyword used to refer to each partition. The result type of the select clause is set<struct(deptname: string, number_of_majors: integer)>. In general, the syntax for the group by clause is

group by f_1: e_1, f_2: e_2, ... , f_k: e_k

where f1: e1, f2: e2, ... , fk: ek is a list of partitioning (grouping) attributes and each partitioning attribute specification fi:ei defines an attribute (field) name fi and an expression ei. The result of applying the grouping (specified in the group by clause) is a set of structures:

set<struct(f_1: t_1, f_2: t_2, . . ., f_k: t_k, partition: bag<B>)>

where ti is the type returned by the expression ei, partition is a distinguished field name (a keyword), and B is a structure whose fields are the iterator variables (s in Q16) declared in the from clause having the appropriate type.

Just as in SQL, a *having* clause can be used to filter the partitioned sets (that is, select only some of the groups based on group conditions). In Q17, the previous query is modified to illustrate the having clause (and also shows the simplified syntax for the select clause). Q17 retrieves for each department having more than 100 majors, the average gpa of its majors. The having clause in Q17 selects only those partitions (groups) that have more than 100 elements (that is, departments with more than 100 students).

```
Q17: select    deptname, avg_gpa: avg (select p.s.gpa from p
                                       in partition)
     from      s in students
     group by  deptname: s.majors_in.dname
     having    count (partition) > 100;
```

Note that the select clause of Q17 returns the average gpa of the students in the partition. The expression

```
select p.s.gpa from p in partition
```

returns a bag of student gpas for that partition. The from clause declares an iterator variable p over the partition collection, which is of type bag<struct(s: Student)>.

Then the path expression `p.s.gpa` is used to access the gpa of each student in the partition.

21.4 Overview of the C++ Language Binding

The C++ language binding specifies how ODL constructs are mapped to C++ constructs. This is done via a C++ class library that provides classes and operations that implement the ODL constructs. An Object Manipulation Language (OML) is needed to specify how database objects are retrieved and manipulated within a C++ program, and this is based on the C++ programming language syntax and semantics. In addition to the ODL/OML bindings, a set of constructs called *physical pragmas* are defined to allow the programmer some control over physical storage issues, such as clustering of objects, utilizing indices, and memory management.

The class library added to C++ for the ODMG standard uses the prefix `d_` for class declarations that deal with database concepts.[26] The goal is that the programmer should think that only one language is being used, not two separate languages. For the programmer to refer to database objects in a program, a class `d_Ref<T>` is defined for each database class `T` in the schema. Hence, program variables of type `d_Ref<T>` can refer to both persistent and transient objects of class `T`.

In order to utilize the various built-in types in the ODMG Object Model such as collection types, various template classes are specified in the library. For example, an abstract class `d_Object<T>` specifies the operations to be inherited by all objects. Similarly, an abstract class `d_Collection<T>` specifies the operations of collections. These classes are not instantiable, but only specify the operations that can be inherited by all objects and by collection objects, respectively. A template class is specified for each type of collection; these include `d_Set<T>`, `d_List<T>`, `d_Bag<T>`, `d_Varray<T>`, and `d_Dictionary<T>`, and correspond to the collection types in the Object Model (see Section 21.1). Hence, the programmer can create classes of types such as `d_Set<d_Ref<Student>>` whose instances would be sets of references to `Student` objects, or `d_Set<String>` whose instances would be sets of Strings. In addition, a class `d_Iterator` corresponds to the Iterator class of the Object Model.

The C++ ODL allows a user to specify the classes of a database schema using the constructs of C++ as well as the constructs provided by the object database library. For specifying the data types of attributes,[27] basic types such as `d_Short` (short integer), `d_UShort` (unsigned short integer), `d_Long` (long integer), and `d_Float` (floating point number) are provided. In addition to the basic data types, several structured literal types are provided to correspond to the structured literal types of the ODMG Object Model. These include `d_String`, `d_Interval`, `d_Date`, `d_Time`, and `d_Timestamp` (see Figure 21.1b).

26. Presumably, `d_` stands for *database* classes.

27. That is, *member variables* in object-oriented programming terminology.

To specify relationships, the keyword `Rel_` is used within the prefix of type names; for example, by writing

```
d_Rel_Ref<Department, _has_majors> majors_in;
```

in the Student class, and

```
d_Rel_Set<Student, _majors_in> has_majors;
```

in the `Department` class, we are declaring that `majors_in` and `has_majors` are relationship properties that are inverses of one another and hence represent a 1:N binary relationship between `Department` and `Student`.

For the OML, the binding overloads the operation *new* so that it can be used to create either persistent or transient objects. To create persistent objects, one must provide the database name and the persistent name of the object. For example, by writing

```
d_Ref<Student> s = new(DB1, 'John_Smith') Student;
```

the programmer creates a named persistent object of type Student in database `DB1` with persistent name `John_Smith`. Another operation, `delete_object()` can be used to delete objects. Object modification is done by the operations (methods) defined in each class by the programmer.

The C++ binding also allows the creation of extents by using the library class `d_Extent`. For example, by writing

```
d_Extent<Person> AllPersons(DB1);
```

the programmer would create a named collection object `AllPersons`—whose type would be `d_Set<Person>`—in the database `DB1` that would hold persistent objects of type `Person`. However, key constraints are not supported in the C++ binding, and any key checks must be programmed in the class methods.[28] Also, the C++ binding does not support persistence via reachability; the object must be statically declared to be persistent at the time it is created.

21.5 Object Database Conceptual Design

Section 21.5.1 discusses how Object Database (ODB) design differs from Relational Database (RDB) design. Section 21.5.2 outlines a mapping algorithm that can be used to create an ODB schema, made of ODMG ODL class definitions, from a conceptual EER schema.

21.5.1 Differences Between Conceptual Design of ODB and RDB

One of the main differences between ODB and RDB design is how relationships are handled. In ODB, relationships are typically handled by having relationship properties or ref-

28. We have only provided a brief overview of the C++ binding. For full details, see Cattell et al. (1997), Ch. 5.

erence attributes that include OID(s) of the related objects. These can be considered as *OID references* to the related objects. Both single references and collections of references are allowed. References for a binary relationship can be declared in a single direction, or in both directions, depending on the types of access expected. If declared in both directions, they may be specified as inverses of one another, thus enforcing the ODB equivalent of the relational referential integrity constraint.

In RDB, relationships among tuples (records) are specified by attributes with matching values. These can be considered as *value references* and are specified via *foreign keys*, which are values of primary key attributes repeated in tuples of the referencing relation. These are limited to being single-valued in each record because multivalued attributes are not permitted in the basic relational model. Thus, M:N relationships must be represented not directly but as a separate relation (table), as discussed in Section 7.1.

Mapping binary relationships that contain attributes is not straightforward in ODBs, since the designer must choose in which direction the attributes should be included. If the attributes are included in both directions, then redundancy in storage will exist and may lead to inconsistent data. Hence, it is sometimes preferable to use the relational approach of creating a separate table by creating a separate class to represent the relationship. This approach can also be used for n-ary relationships, with degree $n > 2$.

Another major area of difference between ODB and RDB design is how inheritance is handled. In ODB, these structures are built into the model, so the mapping is achieved by using the inheritance constructs, such as *derived* (:) and EXTENDS. In relational design, as we discussed in Section 7.2, there are several options to choose from since no built-in construct exists for inheritance in the basic relational model. It is important to note, though, that object-relational and extended-relational systems are adding features to directly model these constructs as well as to include operation specifications in abstract data types (see Chapter 22).

The third major difference is that in ODB design, it is necessary to specify the operations early on in the design since they are part of the class specifications. Although it is important to specify operations during the design phase for all types of databases, it may be delayed in RDB design as it is not strictly required until the implementation phase.

21.5.2 Mapping an EER Schema to an ODB Schema

It is relatively straightforward to design the type declarations of object classes for an ODBMS from an EER schema that contains *neither* categories *nor* n-ary relationships with $n > 2$. However, the operations of classes are not specified in the EER diagram and must be added to the class declarations after the structural mapping is completed. The outline of the mapping from EER to ODL is as follows:

Step 1: Create an ODL *class* for each EER entity type or subclass. The type of the ODL class should include all the attributes of the EER class.[29] *Multivalued attributes* are

29. This implicitly uses a tuple constructor at the top level of the type declaration, but in general, the tuple constructor is not explicitly shown in the ODL class declarations.

declared by using the set, bag, or list constructors.[30] If the values of the multivalued attribute for an object should be ordered, the list constructor is chosen; if duplicates are allowed, the bag constructor should be chosen; otherwise, the set constructor is chosen. *Composite attributes* are mapped into a tuple constructor (by using a struct declaration in ODL).

Declare an extent for each class, and specify any key attributes as keys of the extent. (This is possible only if an extent facility and key constraint declarations are available in the ODBMS.)

Step 2: Add relationship properties or reference attributes for each *binary relationship* into the ODL classes that participate in the relationship. These may be created in one or both directions. If a binary relationship is represented by references in *both* directions, declare the references to be relationship properties that are inverses of one another, if such a facility exists.[31] If a binary relationship is represented by a reference in only *one* direction, declare the reference to be an attribute in the referencing class whose type is the referenced class name.

Depending on the cardinality ratio of the binary relationship, the relationship properties or reference attributes may be single-valued or collection types. They will be single-valued for binary relationships in the 1:1 or *N*:1 directions; they are collection types (set-valued or list-valued[32]) for relationships in the 1:*N* or M:*N* direction. An alternative way for mapping binary M:N relationships is discussed in Step 7 below.

If relationship attributes exist, a tuple constructor (struct) can be used to create a structure of the form `<reference, relationship attributes>`, which may be included instead of the reference attribute. However, this does not allow the use of the inverse constraint. In addition, if this choice is represented in *both directions*, the attribute values will be represented twice, creating redundancy.

Step 3: Include appropriate operations for each class. These are not available from the EER schema and must be added to the database design by referring to the original requirements. A constructor method should include program code that checks any constraints that must hold when a new object is created. A destructor method should check any constraints that may be violated when an object is deleted. Other methods should include any further constraint checks that are relevant.

Step 4: An ODL class that corresponds to a subclass in the EER schema inherits (via EXTENDS) the type and methods of its superclass in the ODL schema. Its *specific* (noninherited) attributes, relationship references, and operations are specified, as discussed in Steps 1, 2, and 3.

---

30. Further analysis of the application domain is needed to decide on which constructor to use because this information is not available from the EER schema.

31. The ODL standard provides for the explicit definition of inverse relationships. Some ODBMS products may not provide this support; in such a case, the programmers must maintain every relationship explicitly by coding the methods that update the objects appropriately.

32. The decision whether to use set or list is not available from the EER schema and must be determined from the requirements.

Step 5: Weak entity types can be mapped in the same way as regular entity types. An alternative mapping is possible for weak entity types that do not participate in any relationships except their identifying relationship; these can be mapped as though they were *composite multivalued attributes* of the owner entity type, by using the `set<struct<...>>` or `list<struct<...>>` constructors. The attributes of the weak entity are included in the `struct<...>` construct, which corresponds to a tuple constructor. Attributes are mapped as discussed in Steps 1 and 2.

Step 6: Categories (union types) in an EER schema are difficult to map to ODL. It is possible to create a mapping similar to the EER-to-relational mapping (see Section 7.2) by declaring a class to represent the category and defining 1:1 relationships between the category and each of its superclasses. Another option is to use a *union type*, if it is available.

Step 7: An n-ary relationship with degree $n > 2$ can be mapped into a separate class, with appropriate references to each participating class. These references are based on mapping a 1:N relationship from each class that represents a participating entity type to the class that represents the n-ary relationship. An M:N binary relationship, especially if it contains relationship attributes, may also use this mapping option, if desired.

The mapping has been applied to a subset of the UNIVERSITY database schema of Figure 4.10 in the context of the ODMG object database standard. The mapped object schema using the ODL notation is shown in Figure 21.6.

21.6 SUMMARY

In this chapter we discussed the proposed standard for object-oriented databases. We started by describing the various constructs of the ODMG object model. The various built-in types, such as Object, Collection, Iterator, Set, List, and so on were described by their interfaces, which specify the built-in operations of each type. These built-in types are the foundation upon which the object definition language (ODL) and object query language (OQL) are based. We also described the difference between objects, which have an ObjectId, and literals, which are values with no OID. Users can declare classes for their application that inherit operations from the appropriate built-in interfaces. Two types of properties can be specified in a user-defined class—attributes and relationships—in addition to the operations that can be applied to objects of the class. The ODL allows users to specify both interfaces and classes, and permits two different types of inheritance—interface inheritance via ":" and class inheritance via EXTENDS. A class can have an extent and keys.

A description of ODL then followed, and an example database schema for the UNIVERSITY database was used to illustrate the ODL constructs. We then presented an overview of the object query language (OQL). The OQL follows the concept of orthogonality in constructing queries, meaning that an operation can be applied to the result of another operation as long as the type of the result is of the correct input type for the operation. The OQL syntax follows many of the constructs of SQL but includes

additional concepts such as path expressions, inheritance, methods, relationships, and collections. Examples of how to use OQL over the UNIVERSITY database were given.

We then gave an overview of the C++ language binding, which extends C++ class declarations with the ODL type constructors but permits seamless integration of C++ with the ODBMS.

Following the description of the ODMG model, we described a general technique for designing object-oriented database schemas. We discussed how object-oriented databases differ from relational databases in three main areas: references to represent relationships, inclusion of operations, and inheritance. We showed how to map a conceptual database design in the EER model to the constructs of object databases.

Review Questions

21.1. What are the differences and similarities between objects and literals in the ODMG Object Model?

21.2. List the basic operations of the following built-in interfaces of the ODMG Object Model: Object, Collection, Iterator, Set, List, Bag, Array, and Dictionary.

21.3. Describe the built-in structured literals of the ODMG Object Model and the operations of each.

21.4. What are the differences and similarities of attribute and relationship properties of a user-defined (atomic) class?

21.5. What are the differences and similarities of EXTENDS and interface ":" inheritance?

21.6. Discuss how persistence is specified in the ODMG Object Model in the C++ binding.

21.7. Why are the concepts of extents and keys important in database applications?

21.8. Describe the following OQL concepts: *database entry points*, *path expressions*, *iterator variables*, *named queries (views)*, *aggregate functions*, *grouping*, and *quantifiers*.

21.9. What is meant by the type orthogonality of OQL?

21.10. Discuss the general principles behind the C++ binding of the ODMG standard.

21.11. What are the main differences between designing a relational database and an object database?

21.12. Describe the steps of the algorithm for object database design by EER-to-OO mapping.

Exercises

21.13. Design an OO schema for a database application that you are interested in. First construct an EER schema for the application; then create the corresponding classes in ODL. Specify a number of methods for each class, and then specify queries in OQL for your database application.

21.14. Consider the AIRPORT database described in Exercise 4.21. Specify a number of operations/methods that you think should be applicable to that application. Specify the ODL classes and methods for the database.

21.15. Map the COMPANY ER schema of Figure 3.2 into ODL classes. Include appropriate methods for each class.

21.16. Specify in OQL the queries in the exercises to Chapters 7 and 8 that apply to the COMPANY database.

Selected Bibliography

Cattell et al. (1997) describes the ODMG 2.0 standard and Cattell et al. (1993) describes the earlier versions of the standard. Several books describe the CORBA architecture—for example, Baker (1996). Other general references to object-oriented databases were given in the bibliographic notes to Chapter 11.

The O2 system is described in Deux et al. (1991) and Bancilhon et al. (1992) includes a list of references to other publications describing various aspects of O2. The O2 model was formalized in Velez et al. (1989). The ObjectStore system is described in Lamb et al. (1991). Fishman et al. (1987) and Wilkinson et al. (1990) discuss IRIS, an object-oriented DBMS developed at Hewlett-Packard laboratories. Maier et al. (1986) and Butterworth et al. (1991) describe the design of GEMSTONE. An OO system supporting open architecture developed at Texas Instruments is described in Thompson et al. (1993). The ODE system developed at ATT Bell Labs is described in Agrawal and Gehani (1989). The ORION system developed at MCC is described in Kim et al. (1990). Morsi et al. (1992) describes an OO testbed.

22

Object-Relational and Extended-Relational Systems

In the preceding chapters we have primarily discussed three data models—the Entity-Relationship (ER) model and its enhanced version, the EER model, in Chapters 3 and 4; the relational data model and its languages and systems in Chapters 5 through 9; and the object-oriented data model and object database languages and standards in Chapters 20 and 21. We discussed how all these data models have been thoroughly developed in terms of the following features:

- Modeling constructs for developing schemas for database applications.
- Constraints facilities for expressing certain types of relationships and constraints on the data as determined by application semantics.
- Operations and language facilities to manipulate the database.

Out of these three models, the ER model and its variations, has been primarily employed in CASE tools that are used for database and software design, whereas the other two models have been used as the basis for commercial DBMSs. This chapter discusses the emerging class of commercial DBMSs that are called *object-relational* or *enhanced relational systems*, and some of the conceptual foundations for these systems. These systems—which are often called object-relational DBMSs (ORDBMSs)—emerged as a way of enhancing the capabilities of relational DBMSs (RDBMSs) with some of the features that appeared in object DBMSs (ODBMSs).

We start in Section 22.1 by giving an overview of the SQL standard, which provides extended and object capabilities to the SQL standard for RDBMS. In Section 22.2 we give a

historical perspective of database technology evolution and current trends to understand why these systems emerged. Section 22.3 gives an overview of the Informix database server as an example of a commercial extended ORDBMS. Section 22.4 discusses the object-relational and extended features of Oracle. Section 22.5 discusses some issues related to the implementation of extended relational systems and Section 22.6 presents an overview of the nested relational model, which provides some of the theoretical foundations behind extending the relational model with complex objects. Section 22.7 is a summary.

Readers interested in typical features of ORDBMS may read Sections 22.1 through 22.4. Other sections may be skipped in an introductory course.

22.1 Overview of SQL and Its Object-Relational Features

We introduced SQL as the standard language for RDBMSs in Chapter 8. As we discussed, SQL was first specified in the 1970s and underwent enhancements in 1989 and 1992. The language continued its evolution toward a new standard called SQL3, which adds object-oriented and other features. A subset of the SQL3 standard, now known as SQL:99, was approved. This section highlights some of the features of SQL3 and SQL:99 with a particular emphasis on the object-relational concepts.

22.1.1 The SQL Standard and Its Components

We will briefly point out what each part of the SQL standard deals with, then describe some SQL features that are relevant to the object extensions to SQL. The SQL standard now includes the following parts:[1]

- SQL/Framework, SQL/Foundation, SQL/Bindings, SQL/Object.
- New parts addressing temporal, transaction aspects of SQL.
- SQL/CLI (Call Level Interface).
- SQL/PSM (Persistent Stored Modules).

SQL/Foundation deals with new data types, new predicates, relational operations, cursors, rules and triggers, user-defined types, transaction capabilities, and stored routines. SQL/CLI (Call Level Interface) (see Chapter 9) provides rules that allow execution of application code without providing source code and avoids the need for preprocessing. It contains about 50 routines for tasks such as connection to the SQL server, allocating and deallocating resources, obtaining diagnostic and implementation information, and controlling termination of transactions. SQL/PSM (Persistent Stored Modules) specifies

1. The discussion about the standard is largely based on Melton and Mattos (1996).

facilities for partitioning an application between a client and a server. The goal is to enhance performance by minimizing network traffic. SQL/Bindings includes Embedded SQL and Direct Invocation. Embedded SQL has been enhanced to include additional exception declarations. SQL/Temporal deals with historical data, time series data, and other temporal extensions, and it is being proposed by the TSQL2 committee.[2] SQL/Transaction specification formalizes the XA interface for use by SQL implementors.

22.1.2 Object-Relational Support in SQL-99

The SQL/Object specification extends SQL-92 to include object-oriented capabilities. We will discuss some of these features by referring to the corresponding object-oriented concepts that we discussed in Chapter 20. The following are some of the features that have been included in SQL-99:

- Some **type constructors** have been added to specify complex objects. These include the *row type*, which corresponds to the tuple (or struct) constructor of Chapter 20. An *array type* for specifying collections is also provided. Other collection type constructors, such as set, list, and bag constructors, are not yet part of the SQL-99 specifications, although some systems include them and they are expected to be in future versions of the standard.
- A mechanism for specifying **object identity** through the use of *reference type* is included.
- **Encapsulation of operations** is provided through the mechanism of user-defined types that may include operations as part of their declaration.
- **Inheritance** mechanisms are provided.

We now discuss each of these concepts in more detail.

Type Constructors. The type constructors *row* and *array* are used to specify complex types. These are also known as **user-defined types**, or UDTs, since the user defines them for a particular application. A **row type** may be specified using the following syntax:

```
CREATE TYPE row_type_name AS [ ROW ] (<component declarations>);
```

The keyword ROW is optional. An example for specifying a row type for addresses and employees may be done as follows:

```
CREATE TYPE Addr_type AS (
  street VARCHAR (45),
  city VARCHAR (25),
  zip CHAR (5)
  );
CREATE TYPE Emp_type AS (
```

2. The full proposal appears in Snodgrass and Jensen (1996). We discuss temporal modeling and introduce TSQL2 in Chapter 23.

```
  name VARCHAR (35),
  addr Addr_type,
  age INTEGER
  );
```

Notice that we can use a previously defined type as a type for an attribute, as illustrated by the addr attribute above. An **array type** may be specified for an attribute whose value will be a collection. For example, suppose that a company has up to ten locations. Then a row type for company may be defined as follows:

```
CREATE TYPE Comp_type AS (
  compname VARCHAR (20),
  location VARCHAR (20) ARRAY [10]
  );
```

Fixed-length array types have their elements referenced using the common notation of square brackets. For example, location[1] refers to the first location value in a location attribute. For row types, the common dot notation is used to refer to components. For example, addr.city refers the the city component of an addr attribute. Currently, array elements cannot be arrays themselves, thus limiting the complexity of the object structures that can be created.

Object Identifiers Using References. A user defined type can be used either as type for an attribute, as illustrated by the addr attribute of Emp_type, or it can be used to specify the row types of tables. For example, we can create two tables based on the row type declarations given earlier as follows:

```
CREATE TABLE Employee OF Emp_type REF IS emp_id SYSTEM GENERATED;
CREATE TABLE Company OF Comp_type (
REF IS comp_id SYSTEM GENERATED,
PRIMARY KEY (compname));
```

The above examples also illustrate how the user can specify that system-generated object identifiers for the individual rows in a table should be created. By using the syntax:

```
REF IS <oid_attribute> <value_generation_method> ;
```

the user declares that the attribute named <oid_attribute> will be used to identify individual tuples in the table. The options for <value_generation_method> are SYSTEM GENERATED or DERIVED. In the former case, the system will automatically generate a unique identifier for each tuple. In the latter case, the traditional method of using the user-provided primary key value to identify tuples is applied.

A component attribute of one tuple may be a **reference** (specified using the keyword REF) to a tuple of another (or possibly the same) table. For example, we can define the following additional row type and correponding table to relate an employee to a company:

```
CREATE TYPE Employment_type AS (
  employee REF (Emp_type) SCOPE (Employee),
  company REF (Comp_type) SCOPE (Company)
);
CREATE TABLE Employment OF Employment_type;
```

The keyword SCOPE specifies the name of the table whose tuples can be referenced by the reference attribute. Notice that this is similar to a foreign key, except that the system generated value is used rather than the primary key value.

SQL uses a **dot notation** to build **path expressions** that refer to the component attributes of tuples and row types. However, for an attribute whose type is REF, the dereferencing symbol -> is used. For example, the query below retrieves employees working in the company named 'ABCXYZ' by querying the `Employment` table:

```
SELECT e.employee->name
FROM Employment AS e
WHERE e.company->compname = 'ABCXYZ';
```

In SQL, -> is used for **dereferencing** and has the same meaning assigned to it in the C programming language. Thus if r is a reference to a tuple and a is a component attribute in that tuple, r -> a is the value of attribute a in that tuple.

Object identifiers can also be explicitly declared in the type definition rather than in the table declaration. For example, the definition of `Emp_type` may be changed as follows:

```
CREATE TYPE Emp_type AS (
  name CHAR (35),
  addr Addr_type,
  age INTEGER,
  emp_id REF (Emp_type)
);
```

In the above example, the `emp_id` values may be specified to be system generated by using the command:

```
CREATE TABLE Employee OF Emp_type
VALUES FOR emp_id ARE SYSTEM GENERATED;
```

If several relations of the same row type exist, SQL provides the SCOPE keyword by which a reference attribute may be made to point to a specific table of that type by using:

```
SCOPE FOR <attribute> IS <relation>
```

Encapsulation of Operations in SQL. In SQL a construct similar to class definition is provided whereby the user can create a named user-defined type with its own behavioral specification by specifying methods (or operations) in addition to the attributes. The general form of an UDT specification with methods is:

```
CREATE TYPE <type-name> (
  list of component attributes with individual types
  declaration of EQUAL and LESS THAN functions
  declaration of other functions (methods)
);
```

For example, suppose we would like to extract the apartment number (if given) from a string that forms the `street` attribute component of the `Addr_type` row type declared previously. We can specify a method for `Addr_type` as follows:

```
CREATE TYPE Addr_type AS (
  street VARCHAR (45),
  city VARCHAR (25),
  zip CHAR (5)
  )
METHOD apt_no() RETURNS CHAR (8);
```

The code for implementing the method still has to be written. We can refer to the method implementation by specifying the file that contains the code for the method as follows:

```
METHOD
CREATE FUNCTION apt_no() RETURNS CHAR (8) FOR Addr_type AS
EXTERNAL NAME '/x/y/aptno.class' LANGUAGE 'java';
```

In this example, the implementation is in the JAVA language, and the code is stored in the specified file path name.

SQL provides certain built-in functions for user defined types. For a UDT called `Type_T`, the **constructor function** `Type_T()` returns a new object of that type. In the new UDT object, every attribute is initialized to its default value. An **observer function** `A` is implicitly created for each attribute A to read its value. Hence, `A(X)` or `X.A` returns the value of attribute A of `Type_T` if X is of type `Type_T`. A **mutator function** for updating an attribute sets the value of the attribute to a new value. SQL allows these functions to be blocked from public use; an EXECUTE privilege is needed to have access to these functions.

In general, a UDT can have a number of user-defined functions associated with it. The syntax is

```
METHOD <name> (<argument_list>) RETURNS <type>;
```

Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended PSM language of SQL (see Chapter 9). External functions are written in a host language, with only their signature (interface) appearing in the UDT definition. An external function definition can be declared as follows:

```
DECLARE EXTERNAL <function_name> <signature>
LANGUAGE <language_name>;
```

Many ORBDMSs have taken the approach of defining a package of Abstract Data Types (ADTs) and associated functions for specific application domains. These could be purchased separately from the basic system. For example, the Data Blades in Informix Universal Server, the Data Cartridges in Oracle, and the Extenders in DB2 can be considered as such packages or libraries of ADTs for specific application domains.

UDTs can be used as the types for attributes in SQL and the parameter types in a function or procedure, and as a source type in a distinct type. **Type Equivalence** is defined in SQL at two levels. Two types are **name equivalent** if and only if they have the same name. Two types are **structurally equivalent** if and only if they have the same number of components and the components are pairwise type equivalent.

Attributes and functions in UDTs are divided into three categories:

- PUBLIC (visible at the UDT interface).
- PRIVATE (not visible at the UDT interface).
- PROTECTED (visible only to subtypes).

It is also possible to define virtual attributes as part of UDTs, which are computed and updated using functions.

Inheritance and Overloading of Functions in SQL. Recall that we already discussed many of the principles of inheritance in Chapter 20. SQL has rules for dealing with inheritance (specified via the UNDER keyword). Associated with inheritance are the rules for overloading of function implementations, and for resolution of function names. These rules can be summarized as follows:

- All attributes are inherited.
- The order of supertypes in the UNDER clause determines the inheritance hierarchy.
- An instance of a subtype can be used in every context in which a supertype instance is used.
- A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
- When a function is called, the best match is selected based on the types of all arguments.
- For dynamic linking, the runtime types of parameters is considered.

Consider the following example to illustrate type inheritance. Suppose that we want to create a subtype Manager_type that inherits all the attributes (and methods) of Emp_type but has an additional attribute dept_managed. Then we can write:

```
CREATE TYPE Manager_type UNDER Emp_type AS (
  dept_managed CHAR (20)
);
```

This inherits all the attributes and methods of the supertype Emp_type, and has an additional specific attribute dept_managed. We could also specify additional specific methods for the subtype.

Another facility in SQL is the supertable/subtable facility, which is similar to the class or extends inheritance discussed in Chapter 20. Here, a subtable inherits every column from its supertable; every row of a subtable corresponds to one and only one row in the supertable; every row in the supertable corresponds to at most one row in a subtable. INSERT, DELETE, and UPDATE operations are appropriately propagated. For example, consider the real_estate_info table defined as follows:

```
CREATE TABLE real_estate_info (
  property real_estate,
  owner CHAR(25),
  price MONEY,
);
```

The following subtables can be defined:

```
CREATE TABLE american_real_estate UNDER real_estate_info;
CREATE TABLE georgia_real_estate UNDER american_real_estate;
CREATE TABLE atlanta_real_estate UNDER georgia_real_estate;
```

In this example, every tuple in the subtable `american_real_estate` must exist in its supertable `real_estate_info`; every tuple in the subtable `georgia_real_estate` must exist in its supertable `american_real _estate`, and so on. However, tuples can exist in a supertable without being in the subtable.

Unstructured Complex Objects in SQL. SQL has new data types for binary large objects (LOBs), and large object locators. Two variations exist for binary large objects (BLOBs) and character large objects (CLOBs). SQL proposes LOB manipulation within the DBMS without having to use external files. Certain operators do not apply to LOB-valued attributes—for example, arithmetic comparisons, group by, and order by. On the other hand, retrieval of partial value, LIKE comparison, concatenation, substring, position, and length are operations that can be applied to LOBs. We will see how large objects are used in Oracle 8.

We have given an overview of the proposed object-oriented facilities in SQL. At this time, both the SQL/Foundations and SQL/Object specification have been standardized. It is evident that the facilities that make SQL object-oriented closely follow what has been implemented in commercial ORDBMSs. SQL/MM (multimedia) is being proposed as a separate standard for multimedia database management with multiple parts: framework, full text, spatial, general purpose facilities, and still image. We will discuss the use of the two-dimensional data types and the image and text Datablades in Informix Universal Server.

22.1.3 Some New Operations and Features in SQL

A major new operation is **linear recursion** for specifying recursive queries. To illustrate this, suppose we have a table called `PART_TABLE(Part1, Part2)`, which contains a tuple <p1, p2> whenever part p1 contains part p2 as a component. A query to produce the **bill of materials** for some part p1 (that is, all component parts needed to produce p1) is written as a recursive query as follows:

```
WITH RECURSIVE
BILL_MATERIAL (Part1, Part2) AS
   (SELECT Part1, Part2
   FROM PART_TABLE
   WHERE Part1 = 'p1'
     UNION ALL
   SELECT PART_TABLE(Part1), PART_TABLE(Part2)
   FROM BILL_MATERIAL, PART_TABLE
   WHERE PART_TABLE.Part1 = BILL_MATERIAL(Part2))
SELECT * FROM BILL_MATERIAL
ORDER BY Part1, Part2;
```

The final result is contained in BILL_MATERIAL(Part1, Part2). The UNION ALL operation is evaluated by taking a union of all tuples generated by the inner block until no new tuples can be generated. Because SQL2 lacks recursion, it was left to the programmer to accomplish it by appropriate iteration.

For security in SQL3, the concept of **role** is introduced, which is similar to a "job description" and is subject to authorization of privileges. The actual persons (user accounts) that are assigned to a role may change, but the role authorization itself does not have to be changed. SQL3 also includes syntax for the specification and use of **triggers** (see Chapter 24) as active rules. Triggering events include the INSERT, DELETE, and UPDATE operations on a table. The trigger can be specified to be considered BEFORE or AFTER the triggering event. The concept of **trigger granularity** is included in SQL3, which allows the specification of both row-level triggers (the trigger is considered for each affected row) or statement-level trigger (the trigger is considered only once for each triggering event).[3] For distributed (client-server) databases (see Chapter 25), the concept of a **client module** is included in SQL3. A client module may contain externally invoked procedures, cursors, and temporary tables, which can be specified using SQL3 syntax.

SQL3 also is being extended with programming language facilities. Routines written in SQL/CLI with full matching of data types and an integrated environment are referred to as **SQL routines.** To make the language computationally complete, the following programming control structures are included in the SQL3 syntax: CALL/RETURN, BEGIN/END, FOR/END_FOR, IF/THEN/ELSE/END_IF, CASE/END_CASE, LOOP/END_LOOP, WHILE/END_WHILE, REPEAT/UNTIL/END_REPEAT, and LEAVE. Variables are declared using DECLARE, and assignments are specified using SET. **External routines** refer to programs written in a host language (ADA, C, COBOL, PASCAL, etc.), possibly containing embedded SQL and having possible type mismatches. The advantage of external routines is that there are existing libraries of such routines that are broadly used, which can cut down a lot of implementation effort for applications. On the other hand, SQL routines are more "pure," but they have not been in wide use. SQL routines can be used for server routines (schema-level routines or modules) or as client modules, and they may be procedures or functions that return values. SQL/CLI is described in Chapter 9.

22.2 Evolution and Current Trends of Database Technology

In the commercial world today, there are several families of DBMS products available. Two important ones are RDBMS and ODBMS, which subscribe to the relational and the object data models respectively. Two other major types of DBMS products—hierarchical and network—are now being referred to as **legacy DBMSs;** these are based on the hierarchical and the network data models, both of which were introduced in the mid-1960s. The hierarchical family primarily has one dominant product—IMS of IBM, whereas the network

3. These concepts are discussed in more detail in Chapter 24.

family includes a large number of DBMSs, such as IDS II (Honeywell), IDMS (Computer Associates), IMAGE (Hewlett Packard), VAX-DBMS (Digital), and TOTAL/SUPRA (Cincom), to name a few. The hierarchical and network data models are summarized in Appendixes E and F.[4]

As database technology evolves, the legacy DBMSs will be gradually replaced by newer offerings. In the interim, we must face the major problem of **interoperability**—the interoperation of a number of databases belonging to all of the disparate families of DBMSs—as well as to legacy file management systems. A whole series of new systems and tools to deal with this problem are emerging as well. More recently, XML has emerged as a new standard for data exchange on the Web (see Chapter 26).

The main forces behind the development of extended ORDBMSs stem from the inability of the legacy DBMSs and the basic relational data model as well as the earlier RDBMSs to meet the challenges of new applications. These are primarily in areas that involve a variety of types of data—for example, text in computer-aided desktop publishing; images in satellite imaging or weather forecasting; complex nonconventional data in engineering designs, in the biological genome information, and in architectural drawings; time series data in history of stock market transactions or sales histories; and spatial and geographic data in maps, air/water pollution data, and traffic data. Hence there is a clear need to design databases that can develop, manipulate, and maintain the complex objects arising from such applications. Furthermore, it is becoming necessary to handle digitized information that represents audio and video data streams (partitioned into individual frames) requiring the storage of BLOBs (binary large objects) in DBMSs.

The popularity of the relational model is helped by a very robust infrastructure in terms of the commercial DBMSs that have been designed to support it. However, the basic relational model and earlier versions of its SQL language proved inadequate to meet the above challenges. Legacy data models like the network data model have a facility to model relationships explicitly, but they suffer from a heavy use of pointers in the implementation and have no concepts like object identity, inheritance, encapsulation, or the support for multiple data types and complex objects. The hierarchical model fits well with some naturally occurring hierarchies in nature and in organizations, but it is too limited and rigid in terms of built-in hierarchical paths in the data. Hence, a trend was started to combine the best features of the object data model and languages into the relational data model so that it can be extended to deal with the challenging applications of today.

In the remainder of this chapter we highlight the features of two representative DBMSs that exemplify the ORDBMS approach: Informix Universal Server and Oracle 8. We conclude by briefly discussing the nested relational model, which has its origin in a series of research proposals and prototype implementations; this provides a theoretical framework of embedding hierarchically structured complex objects within the relational framework.

4. Those chapters devoted to the Network Data Model and the Hierarchical Data Model are available at the Web site for this book.

22.3 THE INFORMIX UNIVERSAL SERVER[5]

The Informix Universal Server is an ORDBMS that combines relational and object database technologies from two previously existing products: Informix and Illustra. The latter system originated from the POSTGRES DBMS, which was a research project at the University of California at Berkeley that was commercialized as the Montage DBMS and went through the name Miro before being named Illustra. Illustra was then acquired by Informix, integrated into its RDBMS, and introduced as the Informix Universal Server—an ORDBMS.

To see why ORDBMSs emerged, we start by focusing on one way of classifying DBMS applications according to two dimensions or axes: (1) complexity of data—the X-dimension—and (2) complexity of querying—the Y-dimension. We can arrange these axes into a simple 0-1 space having four quadrants:

Quadrant 1 (X = 0, Y = 0): Simple data, simple querying

Quadrant 2 (X = 0, Y = 1): Simple data, complex querying

Quadrant 3 (X = 1, Y = 0): Complex data, simple querying

Quadrant 4 (X = 1, Y = 1): Complex data, complex querying

Traditional RDBMSs belong to Quadrant 2. Although they support complex ad hoc queries and updates (as well as transaction processing), they can deal only with simple data that can be modeled as a set of rows in a table. Many object databases (ODBMSs) fall in Quadrant 3, since they concentrate on managing complex data but have somewhat limited querying capabilities based on navigation.[6] In order to move into the fourth quadrant to support both complex data and querying, RDBMSs have been incorporating more complex data objects while ODBMSs have been incorporating more complex querying (for example, the OQL high-level query language, discussed in Chapter 21). The Informix Universal Server belongs to Quadrant 4 because it has extended its basic relational model by incorporating a variety of features that make it object-relational.

Other current ORDBMSs that evolved from RDBMSs include Oracle from Oracle Corporation, Universal DB (UDB) from IBM, Odapter by Hewlett Packard (HP) (which extends Oracle's DBMS), and Open ODB from HP (which extends HP's own Allbase/SQL product). The more successful products seem to be those that maintain the option of working as an RDBMS while introducing the additional functionality. Our intent here is *not* to provide a comparative analysis of these products but only to give an overview of two representative systems.

5. The discussion in this section is primarily based on the book *Object-Relational DBMSs* by Michael Stonebraker and Dorothy Moore (1996), and on the input provided by Magdi Morsi of Informix, Inc. Our discussion may refer to earlier versions of Informix that may not be the most recent.

6. Quadrant 1 includes any software packages that deal with data handling without sophisticated data retrieval and manipulation features. These include spreadsheets like EXCEL, word processors like Microsoft Word, or any file management software.

How Informix Universal Server Extends the Relational Data Model. The extensions to the relational data model provided by Illustra and incorporated into Informix Universal Server fall into the following categories:

- Support for additional or extensible data types.
- Support for user-defined routines (procedures or functions).
- Implicit notion of inheritance.
- Support for indexing extensions.
- Data Blades Application Programming Interface (API).[7]

We give an overview of each of these features in the following sections. We have already introduced in a general way the concepts of data types, type constructors, complex objects, and inheritance in the context of object-oriented models (see Chapter 20).

22.3.1 Extensible Data Types

The architecture of Informix Universal Server comprises the basic DBMS plus a number of **Data Blade modules.** The idea is to treat the DBMS as a razor into which a particular blade is inserted for the support of a specific data type. A number of data types have been provided, including two-dimensional geometric objects (such as points, lines, circles, and ellipses), images, time series, text, and Web pages. When Informix announced the Universal Server, 29 Data Blades were already available.[8] It is also possible for an application to create its own types, thus making the data type notion fully extendible. In addition to the built-in types, Informix Universal Server provides the user with the following four constructs to declare additional types:

1. Opaque type.
2. Distinct type.
3. Row type.
4. Collection type.

When creating a type based on one of the first three options, the user has to provide functions and routines for manipulation and conversion, including built-in, aggregate, and operator functions as well as any additional user-defined functions and routines. The details of these four types are presented in the following sections.

Opaque Type. The opaque type has its internal representation hidden, so it is used for encapsulating a type. The user has to provide casting functions to convert an opaque object between its hidden representation in the server (database) and its visible representation as

7. Data Blades provides extensions to the basic system, as we shall discuss later in Section 22.3.6.

8. For more information on the Data Blades for Informix Universal Server, consult the Web site http://www.informix.com/informix/.

seen by the client (calling program). The user functions *send/receive* are needed to convert to/from the server internal representation from/to the client representation. Similarly, *import/export* functions are used to convert to/from an external representation for bulk copy from/to the internal representation. Several other functions may be defined for processing the opaque types, including *assign()*, *destroy()*, and *compare()*.

The specification of an opaque type includes its name, internal length if fixed, maximum internal length if it is variable length, alignment (which is the byte boundary), as well as whether or not it is hashable (for creating a hash access structure). If we write

```
CREATE OPAQUE TYPE fixed_opaque_udt (INTERNALLENGTH = 8,
   ALIGNMENT = 4, CANNOTHASH);
CREATE OPAQUE TYPE var_opaque_udt (INTERNALLENGTH = variable,
   MAXLEN=1024, ALIGNMENT = 8);
```

then the first statement creates a fixed-length user-defined opaque type, named `fixed_opaque_udt`, and the second statement creates a variable length one, named `var_opaque_udt`. Both are described in an implementation with internal parameters that are not visible to the client.

Distinct Type. The distinct data type is used to extend an existing type through inheritance. The newly defined type inherits the functions/routines of its base type, if they are not overridden. For example, the statement

```
CREATE DISTINCT TYPE hiring_date AS DATE;
```

creates a new user-defined type, `hiring_date`, which can be used like any other built-in type.

Row Type. The row type, which represents a composite attribute, is analogous to a *struct* type in the C programming language.[9] It is a composite type that contains one or more fields. Row type is also used to support inheritance by using the keyword `UNDER`, but the type system supports single inheritance only. By creating tables whose tuples are of a particular row type, it is possible to treat a relation as part of an object-oriented schema and establish inheritance relationships among the relations. In the following row type declarations, `employee_t` and `student_t` inherit (or are *declared under*) `person_t`:

```
CREATE ROW TYPE person_t(name VARCHAR(60), social_security
   NUMERIC(9), birth_date DATE);
CREATE ROW TYPE employee_t(salary NUMERIC(10,2), hired_on
   hiring_date) UNDER person_t;
CREATE ROW TYPE student_t(gpa NUMERIC(4,2), address
   VARCHAR(200)) UNDER person_t;
```

Collection Type. Informix Universal Server collections include lists, sets, and multisets (bags) of built-in types as well as user-defined types.[10] A collection can be the

9. This is similar to the *tuple constructor* discussed in Chapter 20.

10. These are similar to the *collection types* discussed in Chapters 20 and 21.

type of either a field in a row type or a column in a table. The elements of a **set** collection cannot contain duplicate values, and have no specific order. The **list** may contain duplicate elements, and order is significant. Finally, the **multiset** may include duplicates and has no specific order. Consider the following example:

```
CREATE TABLE employee (name VARCHAR(50) NOT NULL, commission
MULTISET (MONEY));
```

Here, the `employee` table contains the commission column, which is of type multiset.

22.3.2 Support for User-Defined Routines

Informix Universal Server supports user-defined functions and routines to manipulate the user defined types. The implementation of these functions can be in either Stored Procedure Language (SPL), or in the C or JAVA programming languages. User-defined functions enable the user to define operator functions such as *plus()*, *minus()*, *times()*, *divide()*, *positive()*, and *negate()*, built-in functions such as cos() and sin(), aggregate functions such as *sum()* and *avg()*, and user-defined routines. This enables Informix Universal Server to handle user-defined types as a built-in type whenever the required functions are defined. The following example specifies an equal function to compare two objects of the `fixed_opaque_udt` type declared earlier:

```
CREATE FUNCTION equal (arg1 fixed_opaque_udt, arg2
   fixed_opaque_udt) RETURNING BOOLEAN;
EXTERNAL NAME "/usr/lib/informix/libopaque.so
   (fixed_opaque_udt_equal)" LANGUAGE C;
END FUNCTION;
```

Informix Universal Server also supports **cast**—a function that converts objects from a source type to a target type. There are two types of user-defined casts: (1) implicit and (2) explicit. Implicit casts are invoked automatically, whereas explicit casts are invoked only when the cast operator is specified explicitly by using "::" or `CAST AS`. If the source and target types have the same internal structure (such as when using the *distinct types* specification), no user-defined functions are needed.

Consider the following example to illustrate explicit casting, where the employee table has a `col1` column of type `var_opaque_udt` and a `col2` column of type `fixed_opaque_udt`.

```
SELECT col1 FROM employee WHERE fixed_opaque_udt::col1 = col2;
```

In order to compare `col1` with `col2`, the cast operator is applied to `col1` to convert it from `var_opaque_udt` to `fixed_opaque_udt`.

22.3.3 Support for Inheritance

Inheritance is addressed at two levels in Informix Universal Server: (1) data (attribute) inheritance and (2) function (operation) inheritance.

Data Inheritance. To create subtypes under existing row types, we use the UNDER keyword as discussed earlier. Consider the following example:

```
CREATE ROW TYPE employee_type (
  ename VARCHAR(25),
  ssn CHAR(9),
  salary INT) ;
CREATE ROW TYPE engineer_type (
  degree VARCHAR(10),
  license VARCHAR(20))
  UNDER employee_type;
CREATE ROW TYPE engr_mgr_type (
  manager_start_date VARCHAR(10),
  dept_managed VARCHAR(20))
  UNDER engineer_type;
```

The above statements create an employee_type and a subtype called engineer_type, which represents employees who are engineers and hence inherits all attributes of employees and has additional properties of degree and license. Another type called engr_mgr_type is a subtype under engineer_type, and hence inherits from engineer_type and implicitly from employee_type as well. Informix Universal Server does not support multiple inheritance. We can now create tables called employee, engineer, and engr_mgr based on these row types.

Note that storage options for storing type hierarchies in tables vary. Informix Universal Server provides the option to store instances in different combinations—for example, one instance (record) at each level or one instance that consolidates all levels—these correspond to the mapping options in Section 7.2. The inherited attributes are either represented repeatedly in the tables at lower levels or are represented with a reference to the object of the supertype. The processing of SQL commands is appropriately modified based on the type hierarchy. For example, the query

```
SELECT *
FROM employee
WHERE salary > 100000;
```

returns the employee information from *all* tables where each selected employee is represented. Thus the scope of the employee table extends to all tuples under employee. As a default, queries on the supertable return columns from the supertable as well as those from the subtables that inherit from that supertable. In contrast, the query

```
SELECT *
FROM ONLY (employee)
WHERE salary > 100000;
```

returns instances from only the employee table because of the keyword ONLY.

It is possible to query a supertable using a *correlation variable* so that the result contains not only supertable_type columns of the subtables but also subtype-specific columns of the subtables. Such a query returns rows of different sizes; the result is called a

jagged row result. Retrieving all information about an employee from all levels in a "jagged form" is accomplished by

```
SELECT e
FROM employee e ;
```

For each employee, depending on whether he or she is an engineer or some other subtype(s), it will return additional sets of attributes from the appropriate subtype tables.

Views defined over supertables cannot be updated because placement of inserted rows is ambiguous.

Function Inheritance. In the same way that data is inherited among tables along a type hierarchy, functions can also be inherited in an ORDBMS. For example, a function overpaid may be defined on `employee_type` to select those employees making a higher salary than Bill Brown as follows:

```
CREATE FUNCTION overpaid (employee_type)
RETURNS BOOLEAN AS
RETURN $1.salary > (SELECT salary
                    FROM employee
                    WHERE ename = 'Bill Brown');
```

The tables under the employee table automatically inherit this function. However, the same function may be redefined for the `engr_mgr_type` as those employees making a higher salary than Jack Jones as follows:

```
CREATE FUNCTION overpaid (engr_mgr_type)
RETURNS BOOLEAN AS
RETURN $1.salary > (SELECT salary
                    FROM employee
                    WHERE ename = 'Jack Jones');
```

For example, consider the query

```
SELECT e.ename
FROM ONLY (employee) e
WHERE overpaid (e);
```

which is evaluated with the first definition of overpaid. The query

```
SELECT g.ename
FROM engineer g
WHERE overpaid (g);
```

also uses the first definition of overpaid (because it was not redefined for engineer), whereas

```
SELECT gm.ename
FROM engr_mgr gm
WHERE overpaid (gm);
```

uses the second definition of overpaid, which overrides the first. This is called **operation (or function) overloading,** as was discussed in Section 20.6 under polymorphism. Note that overpaid—and other functions—can also be treated as *virtual attributes;* hence overpaid may be referenced as `employee.overpaid` or `engr_mgr.overpaid` in a query.

22.3.4 Support for Indexing Extensions

Informix Universal Server supports indexing on user-defined routines on either a single table or a table hierarchy. For example,

```
CREATE INDEX empl_city ON employee (city (address));
```

creates an index on the table employee using the value of the city function.

In order to support user-defined indexes, Informix Universal Server supports operator classes, which are used to support user-defined data types in the generic B-tree as well as other secondary access methods such as R-trees.

22.3.5 Support for External Data Source

Informix Universal Server supports external data sources (such as data stored in a file system) that are mapped to a table in the database called the **virtual table interface.** This interface enables the user to define operations that can be used as *proxies* for the other operations, which are needed to access and manipulate the row or rows associated with the underlying data source. These operations include `open`, `close`, `fetch`, `insert`, and `delete`. Informix Universal Server also supports a set of functions that enables calling SQL statements within a user-defined routine without the overhead of going through a client interface.

22.3.6 Support for Data Blades Application Programming Interface

The Data Blades Application Programming Interface (API) of Informix Universal Server provides new data types and functions for specific types of applications. We will review the extensible data types for two-dimensional operations (required in GIS or CAD applications),[11] the data types related to image storage and management, the time series data type, and a few features of the text data type. The strength of ORDBMSs to deal with the new unconventional applications is largely attributed to these special data types and the tailored functionality that they provide.

Two-Dimensional (Spatial) Data Types. For a two-dimensional application, the relevant data types would include the following:

- A **point** defined by (X, Y) coordinates.
- A **line** defined by its two end points.
- A **polygon** defined by an ordered list of n points that form its vertices.
- A **path** defined by a sequence (ordered list) of points.
- A **circle** defined by its center point and radius.

11. Recall that GIS stands for Geographic Information Systems and CAD for Computer Aided Design.

Given the above as data types, a function such as *distance* may be defined between two points, a point and a line, a line and a circle, and so on, by implementing the appropriate mathematical expressions for distance in a programming language. Similarly, a Boolean *cross* function—which returns true or false depending on whether two geometric objects cross (or intersect)—can be defined between a line and a polygon, a path and a polygon, a line and a circle, and so on. Other relevant Boolean functions for GIS applications would be *overlap* (polygon, polygon), *contains* (polygon, polygon), *contains* (point, polygon), and so on. Note that the concept of overloading (operation polymorphism) applies when the same function name is used with different argument types.

Image Data Types. Images are stored in a variety of standard formats—such as TIFF, GIF, JPEG, photoCD, GROUP 4, and FAX—so one may define a data type for each of these formats and use appropriate library functions to input images from other media or to render images for display. Alternately, IMAGE can be regarded as a single data type with a large number of options for storage of data. The latter option would allow a column in a table to be of type IMAGE and yet accept images in a variety of different formats. The following are some possible functions (operations) on images:

```
rotate (image, angle) returns image.
crop (image, polygon) returns image.
enhance (image) returns image.
```

The *crop* function extracts the portion of an image that intersects with a polygon. The *enhance* function improves the quality of an image by performing contrast enhancement. Multiple images may be supplied as parameters to the following functions:

```
common (image1, image2) returns image.
union (image1, image2) returns image.
similarity (image1, image2) returns number.
```

The *similarity* function typically takes into account the distance between two vectors with components <color, shape, texture, edge> that describe the content of the two images. The VIR Data Blade in Informix Universal Server can be used to accomplish a search on images by content based on the above similarity measure.

Time Series Data Type. Informix Universal Server supports a time series data type that makes the handling of time series data much more simplified than storing it in multiple tables. For example, consider storing the closing stock price on the New York Stock Exchange for more than 3,000 stocks for each workday when the market is open. Such a table can be defined as follows:

```
CREATE TABLE stockprices (
company-name VARCHAR(30),
symbol VARCHAR(5),
prices TIME_SERIES OF FLOAT);
```

Regarding the stock price data for all 3,000 companies over an entire period of, say, several years, only one relation is adequate thanks to the time series data type for the prices attribute. Without this data type, each company would need one table. For example, a table for the coca_cola company (symbol KO) may be declared as follows:

```
CREATE TABLE coca_cola (
recording_date DATE,
price FLOAT);
```

In this table, there would be approximately 260 tuples per year—one for each business day. The time series data type takes into account the calendar, starting time, recording interval (for example, daily, weekly, monthly), and so on. Functions such as extracting a subset of the time series (for example, closing prices during January 1999), summarizing at a coarser granularity (for example, average weekly closing price from the daily closing prices), and constructing moving averages are appropriate.

A query on the stockprices table that gives the moving average for 30 days starting at June 1, 1999 for the `coca_cola` stock can use the `MOVING-AVG` function as follows:

```
SELECT MOVING-AVG(prices, 30, '1999-06-01')
FROM stockprices
WHERE symbol = "KO";
```

The same query in SQL on the table `coca_cola` would be much more complicated to write and would access numerous tuples, whereas the above query on the stockprices table deals with a single row in the table corresponding to this company. It is claimed that using the time series data type provides an order of magnitude performance gain in processing such queries.

Text Data Type. The text DataBlade supports storage, search, and retrieval for text objects. It defines a single data type called **doc,** whose instances are stored as large objects that belong to the built-in data type `large-text`. We will briefly discuss a few important features of this data type.

The underlying storage for `large-text` is the same as that for the `large-object` data type. References to a single large object are recorded in the '`refcount`' system table, which stores information such as number of rows referring to the large object, its OID, its storage manager, its last modification time, and its archive storage manager. Automatic conversion between `large-text` and `text` data types enables any functions with text arguments to be applied to `large-text` objects. Thus concatenation of `large-text` objects as strings as well as extraction of substrings from a `large-text` object are possible.

The Text DataBlade parameters include format for which the default is ASCII, with other possibilities such as `postscript`, `dvipostscript`, `nroff`, `troff`, and `text`. A Text Conversion DataBlade, which is separate from the Text DataBlade, is needed to convert documents among the various formats. An External File parameter instructs the internal representation of doc to store a pointer to an external file rather than copying it to a large object.

For manipulation of doc objects, functions such as the following are used:

```
Import_doc (doc, text) returns doc.
Export_doc (doc, text) returns text.
Assign (doc) returns doc.
Destroy (doc) returns void.
```

The `Assign` and `Destroy` functions already exist for the built-in `large-object` and `large-text` data types, but they must be redefined by the user for objects of type `doc`. The

following statement creates a table called `legaldocuments`, where each row has a title of the document in one column and the document itself as the other column:

```
CREATE TABLE legaldocuments(
title TEXT,
document DOC);
```

To insert a new row into this table of a document called '`lease.contract`,' the following statement can be used:

```
INSERT INTO legaldocuments (title, document)
VALUES ('lease.contract', 'format {troff}:/user/local/
   documents/lease');
```

The second value in the values clause is the path name specifying the file location of this document; the format specification signifies that it is a `troff` document. To search the text, an index must be created, as in the following statement:

```
CREATE INDEX legalindex
ON legaldocuments
USING dtree(document text_ops);
```

In the above, `text_ops` is an op-class (operator class) applicable to an access structure called a `dtree` index, which is a special index structure for documents. When a document of the doc data type is inserted into a table, the text is parsed into individual words. The Text DataBlade is case insensitive; hence, `Housenumber`, `HouseNumber`, or `housenumber` are all considered the same word. Words are *stemmed* according to the WORDNET thesaurus. For example, `houses` or `housing` would be stemmed to `house`, `quickly` to `quick`, and `talked` to `talk`. A **stopword** file is kept, which contains insignificant words such as articles or prepositions that are ignored in the searches. Examples of stopwords include is, not, a, the, but, for, and, if, and so on.

Informix Universal Server provides two sets of routines—the **contains routines** and **text-string functions**—to enable applications to determine which documents contain a certain word or words and which documents are similar. When these functions are used in a search condition, the data is returned in descending order of how well the condition matches the documents, with the best match showing first. There is `WeightContains(index to use, tuple-id of the document, input string)` function and a similar `WeightContainsWords` function that returns a precision number between 0 and 1 indicating the closeness of the match between the input string or input words and the specific document for that tuple-id. To illustrate the use of these functions, consider the following query: Find the titles of legal documents that contain the top ten terms in the document titled '`lease contract`', which can be specified as follows:

```
SELECT d.title
FROM legaldocuments d, legaldocuments l
WHERE contains (d.document, AndTerms (TopNTerms(l.document,10)))
AND l.title = 'lease.contract' AND d.title <> 'lease.contract';
```

This query illustrates how SQL can be enhanced with these data type specific functions to yield a very powerful capability of handing text-related functions. In this query, variable `d` refers to the entire legal corpus whereas `l` refers to the specific document whose title is

'lease.contract'. TopNTerms extracts the top ten terms from the 'lease.contract' document (l); AndTerms combines these terms into a list; and contains compares the terms in that list with the stemwords in every other document (d) in the table legaldocuments.

Summary of Data Blades. As we can see, Data Blades enhance an RDBMS by providing various constructors for abstract data types (ADTs) that allow a user to operate on the data as if it were stored in an ODBMS using the ADTs as classes. This makes the relational system *behave* as an ODBMS, and drastically cuts down the programming effort needed when compared with achieving the same functionality with just SQL embedded in a programming language.

22.4 Object-Relational Features of Oracle 8

In this section we will review a number of features related to the version of the Oracle DBMS product called Release 8.X, which has been enhanced to incorporate object-relational features. Additional features may have been incorporated into subsequent versions of Oracle. A number of additional data types with related manipulation facilities called **cartridges** have been added.[12] For example, the spatial cartridge allows map-based and geographic information to be handled. Management of multimedia data has been facilitated with new data types. Here we highlight the differences between the release 8.X of Oracle (as available at the time of this writing) from the preceding version in terms of the new object-oriented features and data types as well as some storage options. Portions of the language SQL-99, which we discussed in Section 22.1, will be applicable to Oracle. We do not discuss these features here.

22.4.1 Some Examples of Object-Relational Features of Oracle

As an ORDBMS, Oracle 8 continues to provide the capabilities of an RDBMS and additionally supports object-oriented concepts. This provides higher levels of abstraction so that application developers can manipulate application objects as opposed to constructing the objects from relational data. The complex information about an object can be hidden, but the properties (attributes, relationships) and methods (operations) of the object can be identified in the data model. Moreover, object type declarations can be reused via inheritance, thereby reducing application development time and effort. To facilitate object modeling, Oracle introduced the following features (as well as some of the SQL-99 features in Section 22.1).

12. Cartridges in Oracle are somewhat similar to Data Blades in Informix.

Representing Multivalued Attributes Using VARRAY. Some attributes of an object/entity could be multivalued. In the relational model, the multivalued attributes would have to be handled by forming a new table (see Section 7.1 and Section 10.3.2 on first normal form). If ten attributes of a large table were multivalued, we would have eleven tables generated from a single table after normalization. To get the data back, the developer would have to do ten joins across these tables. This does not happen in an object model since all the attributes of an object—including multivalued ones—are encapsulated within the object. Oracle 8 achieves this by using a varying length array (VARRAY) data type, which has the following properties:

1. COUNT: Current number of elements.
2. LIMIT: Maximum number of elements the VARRAY can contain. This is user defined.

Consider the example of a `customer` VARRAY entity with attributes `name` and `phone_numbers`, where `phone_numbers` is multivalued. First, we need to define an object type representing a `phone_number` as follows:

```
CREATE TYPE phone_num_type AS OBJECT (phone_number CHAR(10));
```

Then we define a VARRAY whose elements would be objects of type `phone_num_type`:

```
CREATE TYPE phone_list_type as VARRAY (5) OF phone_num_type;
```

Now we can create the `customer_type` data type as an object with attributes `customer_name` and `phone_numbers`:

```
CREATE TYPE customer_type AS
OBJECT (customer_name VARCHAR(20),
        phone_numbers phone_list_type);
```

It is now possible to create the `customer` table as

```
CREATE TABLE customer OF customer_type;
```

To retrieve a list of all customers and their phone numbers, we can issue a simple query without any joins:

```
SELECT customer_name, phone_numbers
FROM customers;
```

Using Nested Tables to Represent Complex Objects. In object modeling, some attributes of an object could be objects themselves. Oracle 8 accomplishes this by having **nested tables** (see Section 20.6). Here, columns (equivalent to object attributes) can be declared as tables. In the above example let us assume that we have a description attached to every phone number (for example, home, office, cellular). This could be modeled using a nested table by first redefining `phone_num_type` as follows:

```
CREATE TYPE phone_num_type AS
OBJECT (phone_number CHAR(10), description CHAR(30));
```

We next redefine `phone_list_type` as a table of `phone_number_type` as follows:

```
CREATE TYPE phone_list_type AS TABLE OF phone_number_type;
```

We can then create the type `customer_type` and the `customer` table as before. The only difference is that `phone_list_type` is now a nested table instead of a VARRAY. Both structures have similar functions with a few differences. Nested tables do *not* have an upper bound on the number of items whereas VARRAYs do have a limit. Individual items can be retrieved from the nested tables, but this is not possible with VARRAYs. Additional indexes can also be built on nested tables for faster data access.

Object Views. Object views can be used to build virtual objects from relational data, thereby enabling programmers to evolve existing schemas to support objects. This allows relational and object applications to coexist on the same database. In our example, let us say that we had modeled our customer database using a relational model, but management decided to do all future applications in the object model. Moving over to the object view of the same existing relational data would thus facilitate the transition.

22.4.2 Managing Large Objects and Other Storage Features

Oracle can now store extremely large objects like video, audio, and text documents. New data types have been introduced for this purpose. These include the following:

- BLOB (binary large object).
- CLOB (character large object).
- BFILE (binary file stored outside the database).
- NCLOB (fixed-width multibyte CLOB).

All of the above except for BFILE, which is stored outside the database, are stored inside the database along with other data. Only the directory name for a BFILE is stored in the database.

Index Only Tables. Standard Oracle 7.X involves keeping indexes as a B^+-tree that contains pointers to data blocks (see Chapter 14). This gives good performance in most situations. However, both the index and the data block must be accessed to read the data. Moreover, key values are stored twice—in the table and in the index—increasing the storage costs. Oracle 8 supports both the standard indexing scheme and also **index only tables,** where the data records and index are kept together in a B-tree structure (see Chapter 14). This allows faster data retrieval and requires less storage space for small- to medium-sized files where the record size is not too large.

Partitioned Tables and Indexes. Large tables and indexes can be broken down into smaller partitions. The table now becomes a logical structure and the partitions become the actual physical structures that hold the data. This gives the following advantages:

- Continued data availability in the event of partial failures of some partitions.
- Scalable performance allowing substantial growth in data volumes.
- Overall performance improvement in query and transaction processing.

22.5 Implementation and Related Issues for Extended Type Systems

There are various implementation issues regarding the support of an extended type system with associated functions (operations). We briefly summarize them here.[13]

- The ORDBMS must dynamically link a user-defined function in its address space only when it is required. As we saw in the case of the two ORDBMSs, numerous functions are required to operate on two- or three-dimensional spatial data, images, text, and so on. With a static linking of all function libraries, the DBMS address space may increase by an order of magnitude. Dynamic linking is available in the two ORDBMSs that we studied.
- Client-server issues deal with the placement and activation of functions. If the server needs to perform a function, it is best to do so in the DBMS address space rather than remotely, due to the large amount of overhead. If the function demands computation that is too intensive or if the server is attending to a very large number of clients, the server may ship the function to a separate client machine. For security reasons, it is better to run functions at the client using the user ID of the client. In the future functions are likely to be written in interpreted languages like JAVA.
- It should be possible to run queries inside functions. A function must operate the same way whether it is used from an application using the application program interface (API), or whether it is invoked by the DBMS as a part of executing SQL with the function embedded in an SQL statement. Systems should support a nesting of these "callbacks."
- Because of the variety in the data types in an ORDBMS and associated operators, efficient storage and access of the data is important. For spatial data or multidimensional data, new storage structures such as R-trees, quad trees, or Grid files may be used. The ORDBMS must allow new types to be defined with new access structures. Dealing with large text strings or binary files also opens up a number of storage and search options. It should be possible to explore such new options by defining new data types within the ORDBMS.

Other Issues Concerning Object-Relational Systems. In the above discussion of Informix Universal Server and Oracle 8, we have concentrated on how an ORDBMS extends the relational model. We discussed the features and facilities it provides to operate on relational data stored as tables as if it were an object database. There are other obvious problems to consider in the context of an ORDBMS:

- *Object-relational database design:* We described a procedure for designing object schemas in Section 21.5. Object-relational design is more complicated because we have to consider not only the underlying design considerations of application semantics and dependencies in the relational data model (which we discussed in Chapters 10

13. This discussion is derived largely from Stonebraker and Moore (1996).

and 11) but also the object-oriented nature of the extended features that we have just discussed.

- *Query processing and optimization:* By extending SQL with functions and rules, this problem is further compounded beyond the query optimization overview that we discuss for the relational model in Chapter 15.
- *Interaction of rules with transactions:* Rule processing as implied in SQL covers more than just the update-update rules (see Section 24.1), which are implemented in RDBMSs as triggers. Moreover, RDBMSs currently implement only immediate execution of triggers. A deferred execution of triggers involves additional processing.

22.6 The Nested Relational Model

To complete this discussion, we summarize in this section an approach that proposes the use of nested tables, also known as nonnormal form relations. No commercial DBMS has chosen to implement this concept in its original form. The **nested relational model** removes the restriction of first normal form (1NF, see Chapter 11) from the basic relational model, and thus is also known as the **Non-1NF** or **Non-First Normal Form (NFNF)** or **NF^2** relational model. In the basic relational model—also called the **flat relational model**—attributes are required to be single-valued and to have atomic domains. The nested relational model allows composite and multivalued attributes, thus leading to complex tuples with a hierarchical structure. This is useful for representing objects that are naturally hierarchically structured. In Figure 22.1, part (a) shows a nested relation schema DEPT based on part of the COMPANY database, and part (b) gives an example of a Non-1NF tuple in DEPT.

To define the DEPT schema as a nested structure, we can write the following:

```
dept = (dno, dname, manager, employees, projects, locations)
employees = (ename, dependents)
projects = (pname, ploc)
locations = (dloc)
dependents = (dname, age)
```

First, all attributes of the DEPT relation are defined. Next, any nested attributes of DEPT—namely, EMPLOYEES, PROJECTS, and LOCATIONS—are themselves defined. Next, any second-level nested attributes, such as DEPENDENTS of EMPLOYEES, are defined, and so on. All attribute names must be distinct in the nested relation definition. Notice that a nested attribute is typically a **multivalued composite attribute,** thus leading to a "nested relation" *within each tuple*. For example, the value of the PROJECTS attribute within each DEPT tuple is a relation with two attributes (PNAME, PLOC). In the DEPT tuple of Figure 22.1b, the PROJECTS attribute contains three tuples as its value. Other nested attributes may be **multivalued simple attributes,** such as LOCATIONS of DEPT. It is also possible to have a nested attribute that is **single-valued and composite,** although most nested relational models treat such an attribute as though it were multivalued.

(a)

DNO	DNAME	MANAGER	EMPLOYEES			PROJECTS		LOCATIONS
			ENAME	DEPENDENTS		PNAME	PLOC	DLOC
				DNAME	AGE			

(b)

DNO	DNAME	MANAGER	ENAME	DEPENDENTS		PNAME	PLOC	DLOC
				DNAME	AGE			
4	Administration	Wallace	Zelaya	Thomas	8	New benefits	Stafford	Stafford
				Jennifer	6	Computerization	Stafford	Greenway
			Wallace	Jack	18	Phone System	Greenway	
				Robert	15			
				Mary	10			
			Jabbar					

(c)

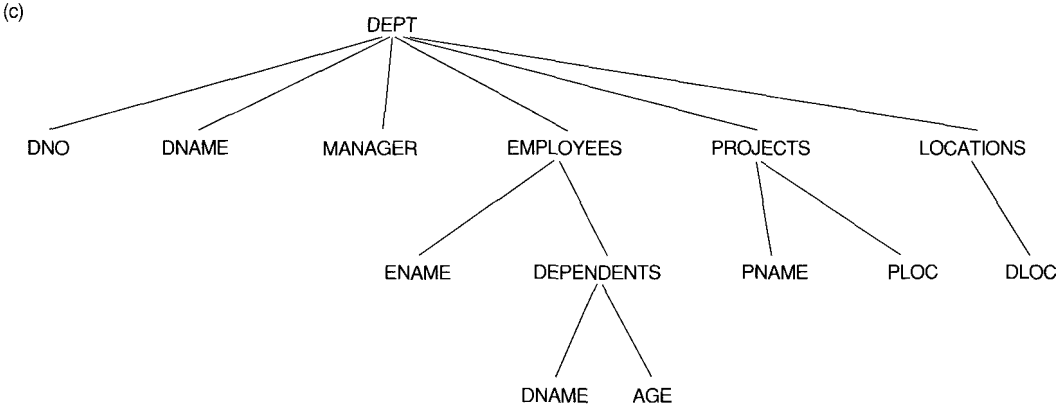


FIGURE 22.1 Illustrating a nested relation. (a) DEPT schema. (b) Example of a Non-1NF tuple of DEPT. (c) Tree representation of DEPT schema.

When a nested relational database schema is defined, it consists of a number of external relation schemas; these define the top level of the individual nested relations. In addition, nested attributes are called **internal relation schemas,** since they define relational structures that are nested inside another relation. In our example, DEPT is the only external relation. All the others—EMPLOYEES, PROJECTS, LOCATIONS, and DEPENDENTS—are internal relations. Finally, **simple attributes** appear at the leaf level and are not nested.

We can represent each relation schema by means of a tree structure, as shown in Figure 22.1c, where the root is an external relation schema, the leaves are simple attributes, and the internal nodes are internal relation schemas. Notice the similarity between this representation and a hierarchical schema (see Appendix E) and XML (see Chapter 26).

It is important to be aware that the three first-level nested relations in DEPT represent *independent information*. Hence, EMPLOYEES represents the employees *working for* the department, PROJECTS represents the projects *controlled by* the department, and LOCATIONS represents the various department locations. The relationship between EMPLOYEES and PROJECTS is not represented in the schema; this is an M:N relationship, which is difficult to represent in a hierarchical structure.

Extensions to the relational algebra and to the relational calculus, as well as to SQL, have been proposed for nested relations. The interested reader is referred to the selected bibliography at the end of this chapter for details. Here, we illustrate two operations, **NEST** and **UNNEST,** that can be used to augment standard relational algebra operations for converting between nested and flat relations. Consider the flat EMP_PROJ relation of Figure 11.4, and suppose that we project it over the attributes SSN, PNUMBER, HOURS, ENAME as follows:

$$\text{EMP\_PROJ\_FLAT} \leftarrow \pi_{\text{SSN, ENAME, PNUMBER, HOURS}}(\text{EMP\_PROJ})$$

To create a nested version of this relation, where one tuple exists for each employee and the (PNUMBER, HOURS) are nested, we use the NEST operation as follows:

$$\text{EMP\_PROJ\_NESTED} \leftarrow \text{NEST}_{\text{PROJS = (PNUMBER, HOURS)}}(\text{EMP\_PROJ\_FLAT})$$

The effect of this operation is to create an internal nested relation PROJS = (PNUMBER, HOURS) within the external relation EMP_PROJ_NESTED. Hence, NEST groups together the tuples *with the same value* for the attributes that are *not specified* in the NEST operation; these are the SSN and ENAME attributes in our example. For each such group, which represents one employee in our example, a single nested tuple is created with an internal nested relation PROJS = (PNUMBER, HOURS). Hence, the EMP_PROJ_NESTED relation looks like the EMP_PROJ relation shown in Figure 11.9a and b.

Notice the similarity between nesting and grouping for aggregate functions. In the former, each group of tuples becomes a single nested tuple; in the latter, each group becomes a single summary tuple after an aggregate function is applied to the group.

The UNNEST operation is the inverse of NEST. We can reconvert EMP_PROJ_NESTED to EMP_PROJ_FLAT as follows:

$$\text{EMP\_PROJ\_FLAT} \leftarrow \text{UNNEST}_{\text{PROJS = (PNUMBER, HOURS)}}(\text{EMP\_PROJ\_NESTED})$$

Here, the PROJS nested attribute is flattened into its components PNUMBER, HOURS.

22.7 SUMMARY

In this chapter, we first gave an overview of the object-oriented features in SQL-99, which are applicable to object-relational systems. Then we discussed the history and current trends in database management systems that led to the development of object-relational DBMSs (ORDBMSs). We then focused on some of the features of Informix Universal Server

and of Oracle 8 in order to illustrate how commercial RDBMSs are being extended with object features. Other commercial RDBMSs are providing similar extensions. We saw that these systems also provide Data Blades (Informix) or Cartridges (Oracle) that provide specific type extensions for newer application domains, such as spatial, time series, or text/document databases. Because of the extendibility of ORDBMSs, these packages can be included as abstract data type (ADT) libraries whenever the users need to implement the types of applications they support. Users can also implement their own extensions as needed by using the ADT facilities of these systems. We briefly discussed some implementation issues for ADTs. Finally, we gave an overview of the nested relational model, which extends the flat relational model with hierarchically structured complex objects.

Selected Bibliography

The references provided for the object-oriented database approach in Chapters 11 and 12 are also relevant for object-relational systems. Stonebraker and Moore (1996) provides a comprehensive reference for object-relational DBMSs. The discussion about concepts related to Illustra in that book are mostly applicable to the current Informix Universal Server. Kim (1995) discusses many issues related to modern database systems that include object orientation. For the most current information on Informix and Oracle, consult their Web sites: *www.informix.com* and *www.oracle.com*, respectively.

The SQL3 standard is described in various publications of the ISO WG3 (Working Group 3) reports; for example, see Kulkarni et al. (1995) and Melton et al. (1991). An excellent tutorial on SQL3 was given at the Very Large Data Bases Conference by Melton and Mattos (1996). Ullman and Widom (1997) have a good discussion of SQL3 with examples.

For issues related to rules and triggers, Widom and Ceri (1995) have a collection of chapters on active databases. Some comparative studies—for example, Ketabchi et al. (1990)—compare relational DBMSs with object DBMSs; their conclusion shows the superiority of the object-oriented approach for nonconventional applications. The nested relational model is discussed in Schek and Scholl (1985), Jaeshke and Schek (1982), Chen and Kambayashi (1991), and Makinouchi (1977), among others. Algebras and query languages for nested relations are presented in Paredaens and VanGucht (1992), Pistor and Andersen (1986), Roth et al. (1988), and Ozsoyoglu et al. (1987), among others. Implementation of prototype nested relational systems is described in Dadam et al. (1986), Deshpande and VanGucht (1988), and Schek and Scholl (1989).