Part 7

Object DBMSs

Chapter 25 Introduction to Object DBMSs

Chapter Objectives

In this chapter you will learn:

- The requirements for advanced database applications.
- Why relational DBMSs currently are not well suited to supporting advanced database applications.
- The concepts associated with object-orientation:
  - abstraction, encapsulation, and information hiding;
  - objects and attributes;
  - object identity;
  - methods and messages;
  - classes, subclasses, superclasses, and inheritance;
  - overloading;
  - polymorphism and dynamic binding.
- The problems associated with storing objects in a relational database.
- What constitutes the next generation of database systems.
- The basics of object-oriented database analysis and design with UML.

Object-orientation is an approach to software construction that has shown considerable promise for solving some of the classic problems of software development. The underlying concept behind object technology is that all software should be constructed out of standard, reusable components wherever possible. Traditionally, software engineering and database management have existed as separate disciplines. Database technology has concentrated on the static aspects of information storage, while software engineering has modeled the dynamic aspects of software. With the arrival of the third generation of Database Management Systems, namely **Object-Oriented Database Management Systems** (OODBMSs) and **Object-Relational Database Management Systems** (ORDBMSs), the two disciplines have been combined to allow the concurrent modeling of both data and the processes acting upon the data.

However, there is currently significant dispute regarding this next generation of DBMSs. The success of relational systems in the past two decades is evident, and the

traditionalists believe that it is sufficient to extend the relational model with additional (object-oriented) capabilities. Others believe that an underlying relational model is inadequate to handle complex applications, such as computer-aided design, computer-aided software engineering, and geographic information systems. To help understand these new types of DBMS, and the arguments on both sides, we devote four chapters to discussing the technology and issues behind them.

In Chapter 26 we consider the emergence of OODBMSs and examine some of the issues underlying these systems. In Chapter 27 we examine the object model proposed by the Object Data Management Group (ODMG), which has become a *de facto* standard for OODBMSs, and ObjectStore, a commercial OODBMS. In Chapter 28 we consider the emergence of ORDBMSs and examine some of the issues underlying these systems. In particular, we will examine SQL:2003, the latest release of the ANSI/ISO standard for SQL, and examine some of the object-oriented features of Oracle. In this chapter we discuss concepts that are common to both OODBMSs and ORDBMSs.

Structure of this Chapter

In Section 25.1 we examine the requirements for the advanced types of database applications that are becoming more commonplace, and in Section 25.2 we discuss why traditional RDBMSs are not well suited to supporting these new applications. In Section 25.3 we provide an introduction to the main object-oriented concepts and in Section 25.4 we examine the problems associated with storing objects in a relational database. In Section 25.5 we provide a brief history of database management systems leading to their third generation, namely object-oriented and object-relational DBMSs. In Section 25.6 we briefly examine how the methodology for conceptual and logical database design presented in Chapters 15 and 16 can be extended to handle object-oriented database design. The examples in this chapter are once again drawn from the *DreamHome* case study documented in Section 10.4 and Appendix A.

25.1 Advanced Database Applications

The computer industry has seen significant changes in the last decade. In database systems, we have seen the widespread acceptance of RDBMSs for traditional business applications, such as order processing, inventory control, banking, and airline reservations. However, existing RDBMSs have proven inadequate for applications whose needs are quite different from those of traditional business database applications. These applications include:

- computer-aided design (CAD);
- computer-aided manufacturing (CAM);
- computer-aided software engineering (CASE);
- network management systems;
- office information systems (OIS) and multimedia systems;

- digital publishing;
- geographic information systems (GIS);
- interactive and dynamic Web sites.

Computer-aided design (CAD)

A CAD database stores data relating to mechanical and electrical design covering, for example, buildings, aircraft, and integrated circuit chips. Designs of this type have some common characteristics:

- Design data is characterized by a large number of types, each with a small number of instances. Conventional databases are typically the opposite. For example, the *DreamHome* database consists of only a dozen or so relations, although relations such as PropertyForRent, Client, and Viewing may contain thousands of tuples.
- Designs may be very large, perhaps consisting of millions of parts, often with many interdependent subsystem designs.
- The design is not static but evolves through time. When a design change occurs, its implications must be propagated through all design representations. The dynamic nature of design may mean that some actions cannot be foreseen at the beginning.
- Updates are far-reaching because of topological or functional relationships, tolerances, and so on. One change is likely to affect a large number of design objects.
- Often, many design alternatives are being considered for each component, and the correct version for each part must be maintained. This involves some form of version control and configuration management.
- There may be hundreds of staff involved with the design, and they may work in parallel on multiple versions of a large design. Even so, the end-product must be consistent and coordinated. This is sometimes referred to as *cooperative engineering*.

Computer-aided manufacturing (CAM)

A CAM database stores similar data to a CAD system, in addition to data relating to discrete production (such as cars on an assembly line) and continuous production (such as chemical synthesis). For example, in chemical manufacturing there will be applications that monitor information about the state of the system, such as reactor vessel temperatures, flow rates, and yields. There will also be applications that control various physical processes, such as opening valves, applying more heat to reactor vessels, and increasing the flow of cooling systems. These applications are often organized in a hierarchy, with a top-level application monitoring the entire factory and lower-level applications monitoring individual manufacturing processes. These applications must respond in real time and be capable of adjusting processes to maintain optimum performance within tight tolerances. The applications use a combination of standard algorithms and custom rules to respond to different conditions. Operators may modify these rules occasionally to optimize performance based on complex historical data that the system has to maintain. In this example, the system has to maintain large volumes of data that is hierarchical in nature and maintain complex relationships between the data. It must also be able to rapidly navigate the data to review and respond to changes.

Computer-aided software engineering (CASE)

A CASE database stores data relating to the stages of the software development lifecycle: planning, requirements collection and analysis, design, implementation, testing, maintenance, and documentation. As with CAD, designs may be extremely large, and cooperative engineering is the norm. For example, software configuration management tools allow concurrent sharing of project design, code, and documentation. They also track the dependencies between these components and assist with change management. Project management tools facilitate the coordination of various project management activities, such as the scheduling of potentially highly complex interdependent tasks, cost estimation, and progress monitoring.

Network management systems

Network management systems coordinate the delivery of communication services across a computer network. These systems perform such tasks as network path management, problem management, and network planning. As with the chemical manufacturing example we discussed earlier, these systems also handle complex data and require real-time performance and continuous operation. For example, a telephone call might involve a chain of network switching devices that route a message from sender to receiver, such as:

$$\text{Node} \Leftrightarrow \text{Link} \Leftrightarrow \text{Node} \Leftrightarrow \text{Link} \Leftrightarrow \text{Node} \Leftrightarrow \text{Link} \Leftrightarrow \text{Node}$$

where each Node represents a port on a network device and each Link represents a slice of bandwidth reserved for that connection. However, a node may participate in several different connections and any database that is created has to manage a complex graph of relationships. To route connections, diagnose problems, and balance loadings, the network management systems have to be capable of moving through this complex graph in real time.

Office information systems (OIS) and multimedia systems

An OIS database stores data relating to the computer control of information in a business, including electronic mail, documents, invoices, and so on. To provide better support for this area, we need to handle a wider range of data types other than names, addresses, dates, and money. Modern systems now handle free-form text, photographs, diagrams, and audio and video sequences. For example, a multimedia document may handle text, photographs, spreadsheets, and voice commentary. The documents may have a specific structure imposed on them, perhaps described using a mark-up language such as SGML (Standardized Generalized Markup Language), HTML (HyperText Markup Language), or XML (eXtended Markup Language), as we discuss in Chapter 30.

Documents may be shared among many users using systems such as electronic mail and bulletin-boards based on Internet technology.[†] Again, such applications need to store data that has a much richer structure than tuples consisting of numbers and text strings. There is also an increasing need to capture handwritten notes using electronic devices. Although

[†] A potentially damaging criticism of database systems, as noted by a number of observers, is that the largest 'database' in the world – the World Wide Web – initially developed with little or no use of database technology. We discuss the integration of the World Wide Web and DBMSs in Chapter 29.

many notes can be transcribed into ASCII text using handwriting analysis techniques, most such data cannot. In addition to words, handwritten data can include sketches, diagrams, and so on.

In the *DreamHome* case study, we may find the following requirements for handling multimedia.

- *Image data* A client may query an image database of properties for rent. Some queries may simply use a textual description to identify images of desirable properties. In other cases it may be useful for the client to query using graphical images of the features that may be found in desirable properties (such as bay windows, internal cornicing, or roof gardens).
- *Video data* A client may query a video database of properties for rent. Again, some queries may simply use a textual description to identify the video images of desirable properties. In other cases it may be useful for the client to query using video features of the desired properties (such as views of the sea or surrounding hills).
- *Audio data* A client may query an audio database that describes the features of properties for rent. Once again, some queries may simply use a textual description to identify the desired property. In other cases it may be useful for the client to use audio features of the desired properties (such as the noise level from nearby traffic).
- *Handwritten data* A member of staff may create notes while carrying out inspections of properties for rent. At a later date, he or she may wish to query such data to find all notes made about a flat in Novar Drive with dry rot.

Digital publishing

The publishing industry is likely to undergo profound changes in business practices over the next decade. It is becoming possible to store books, journals, papers, and articles electronically and deliver them over high-speed networks to consumers. As with office information systems, digital publishing is being extended to handle multimedia documents consisting of text, audio, image, and video data and animation. In some cases, the amount of information available to be put online is enormous, in the order of petabytes (10^{15} bytes), which would make them the largest databases that a DBMS has ever had to manage.

Geographic information systems (GIS)

A GIS database stores various types of spatial and temporal information, such as that used in land management and underwater exploration. Much of the data in these systems is derived from survey and satellite photographs, and tends to be very large. Searches may involve identifying features based, for example, on shape, color, or texture, using advanced pattern-recognition techniques.

For example, EOS (Earth Observing System) is a collection of satellites launched by NASA in the 1990s to gather information that will support scientists concerned with long-term trends regarding the earth's atmosphere, oceans, and land. It is anticipated that these satellites will return over one-third of a petabyte of information per year. This data will be integrated with other data sources and will be stored in EOSDIS (EOS Data and Information System). EOSDIS will supply the information needs of both scientists and

non-scientists. For example, schoolchildren will be able to access EOSDIS to see a simulation of world weather patterns. The immense size of this database and the need to support thousands of users with very heavy volumes of information requests will provide many challenges for DBMSs.

Interactive and dynamic Web sites

Consider a Web site that has an online catalog for selling clothes. The Web site maintains a set of preferences for previous visitors to the site and allows a visitor to:

- browse through thumbnail images of the items in the catalog and select one to obtain a full-size image with supporting details;
- search for items that match a user-defined set of criteria;
- obtain a 3D rendering of any item of clothing based on a customized specification (for example, color, size, fabric);
- modify the rendering to account for movement, illumination, backdrop, occasion, and so on;
- select accessories to go with the outfit, from items presented in a sidebar;
- select a voiceover commentary giving additional details of the item;
- view a running total of the bill, with appropriate discounts;
- conclude the purchase through a secure online transaction.

The requirements for this type of application are not that different from some of the above advanced applications: there is a need to handle multimedia content (text, audio, image, video data, and animation) and to interactively modify the display based on user preferences and user selections. As well as handling complex data, the site also has the added complexity of providing 3D rendering. It is argued that in such a situation the database is not just presenting information to the visitor but is *actively* engaged in selling, dynamically providing customized information and atmosphere to the visitor (King, 1997).

As we discuss in Chapters 29 and 30, the Web now provides a relatively new paradigm for data management, and languages such as XML hold significant promise, particularly for the e-Commerce market. The Forrester Research Group is predicting that business-to-business transactions will reach US$2.1 trillion in Europe and US$7 trillion in the US by 2006. Overall, e-Commerce is expected to account for US$12.8 trillion in worldwide corporate revenue by 2006 and potentially represent 18% of sales in the global economy. As the use of the Internet increases and the technology becomes more sophisticated, then we will see Web sites and business-to-business transactions handle much more complex and interrelated data.

Other advanced database applications include:

- *Scientific and medical applications*, which may store complex data representing systems such as molecular models for synthetic chemical compounds and genetic material.
- *Expert systems*, which may store knowledge and rule bases for artificial intelligence (AI) applications.
- Other applications with complex and interrelated objects and procedural data.

25.2 Weaknesses of RDBMSs

In Chapter 3 we discussed how the relational model has a strong theoretical foundation, based on first-order predicate logic. This theory supported the development of SQL, a declarative language that has now become the standard language for defining and manipulating relational databases. Other strengths of the relational model are its simplicity, its suitability for Online Transaction Processing (OLTP), and its support for data independence. However, the relational data model, and relational DBMSs in particular, are not without their disadvantages. Table 25.1 lists some of the more significant weaknesses often cited by the proponents of the object-oriented approach. We discuss these weaknesses in this section and leave readers to judge for themselves the applicability of these weaknesses.

Poor representation of 'real world' entities

The process of normalization generally leads to the creation of relations that do not correspond to entities in the 'real world'. The fragmentation of a 'real world' entity into many relations, with a physical representation that reflects this structure, is inefficient leading to many joins during query processing. As we have already seen in Chapter 21, the join is one of the most costly operations to perform.

Semantic overloading

The relational model has only one construct for representing data and relationships between data, namely the *relation*. For example, to represent a many-to-many (*:*) relationship between two entities A and B, we create three relations, one to represent each of the entities A and B, and one to represent the relationship. There is no mechanism to distinguish between entities and relationships, or to distinguish between different kinds of relationship that exist between entities. For example, a 1:* relationship might be *Has*, *Owns*, *Manages*, and so on. If such distinctions could be made, then it might be possible to

Table 25.1 Summary of weaknesses of relational DBMSs.

Weakness
Poor representation of 'real world' entities
Semantic overloading
Poor support for integrity and enterprise constraints
Homogeneous data structure
Limited operations
Difficulty handling recursive queries
Impedance mismatch
Other problems with RDBMSs associated with concurrency, schema changes, and poor navigational access

build the semantics into the operations. It is said that the relational model is **semantically overloaded**.

There have been many attempts to overcome this problem using **semantic data models**, that is, models that represent more of the meaning of data. The interested reader is referred to the survey papers by Hull and King (1987) and Peckham and Maryanski (1988). However, the relational model is not completely without semantic features. For example, it has domains and keys (see Section 3.2), and functional, multi-valued, and join dependencies (see Chapters 13 and 14).

Poor support for integrity and general constraints

Integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate. In Section 3.3 we introduced the concepts of entity and referential integrity, and in Section 3.2.1 we introduced domains, which are also a type of constraint. Unfortunately, many commercial systems do not fully support these constraints and it is necessary to build them into the applications. This, of course, is dangerous and can lead to duplication of effort and, worse still, inconsistencies. Furthermore, there is no support for general constraints in the relational model, which again means they have to be built into the DBMS or the application.

As we have seen in Chapters 5 and 6, the SQL standard helps partially resolve this claimed deficiency by allowing some types of constraints to be specified as part of the Data Definition Language (DDL).

Homogeneous data structure

The relational model assumes both horizontal and vertical homogeneity. Horizontal homogeneity means that each tuple of a relation must be composed of the same attributes. Vertical homogeneity means that the values in a particular column of a relation must all come from the same domain. Further, the intersection of a row and column must be an atomic value. This fixed structure is too restrictive for many 'real world' objects that have a complex structure, and it leads to unnatural joins, which are inefficient as mentioned above. In defense of the relational data model, it could equally be argued that its symmetric structure is one of the model's strengths.

Among the classic examples of complex data and interrelated relationships is a parts explosion where we wish to represent some object, such as an aircraft, as being composed of parts and composite parts, which in turn are composed of other parts and composite parts, and so on. This weakness has led to research in complex object or non-first normal form (NF^2) database systems, addressed in the papers by, for example, Jaeschke and Schek (1982) and Bancilhon and Khoshafian (1989). In the latter paper, objects are defined recursively as follows:

(1) Every atomic value (such as integer, float, string) is an object.

(2) If $a_1, a_2, \ldots, a_n$ are distinct attribute names and $o_1, o_2, \ldots, o_n$ are objects, then $[a_1{:}o_1, a_2{:}o_2, \ldots, a_n{:}o_n]$ is a tuple object.

(3) If $o_1, o_2, \ldots, o_n$ are objects, then $S = \{o_1, o_2, \ldots, o_n\}$ is a set object.

In this model, the following would be valid objects:

Atomic objects	B003, John, Glasgow
Set	{SG37, SG14, SG5}
Tuple	[branchNo: B003, street: 163 Main St, city: Glasgow]
Hierarchical tuple	[branchNo: B003, street: 163 Main St, city: Glasgow, staff: {SG37, SG14, SG5}]
Set of tuples	{[branchNo: B003, street: 163 Main St, city: Glasgow], [branchNo: B005, street: 22 Deer Rd, city: London]}
Nested relation	{[branchNo: B003, street: 163 Main St, city: Glasgow, staff: {SG37, SG14, SG5}], [branchNo: B005, street: 22 Deer Rd, city: London, staff: {SL21, SL41}]}

Many RDBMSs now allow the storage of **Binary Large Objects** (BLOBs). A BLOB is a data value that contains binary information representing an image, a digitized video or audio sequence, a procedure, or any large unstructured object. The DBMS does not have any knowledge concerning the content of the BLOB or its internal structure. This prevents the DBMS from performing queries and operations on inherently rich and structured data types. Typically, the database does not manage this information directly, but simply contains a reference to a file. The use of BLOBs is not an elegant solution and storing this information in external files denies it many of the protections naturally afforded by the DBMS. More importantly, BLOBs cannot contain other BLOBs, so they cannot take the form of composite objects. Further, BLOBs generally ignore the behavioral aspects of objects. For example, a picture can be stored as a BLOB in some relational DBMSs. However, the picture can only be stored and displayed. It is not possible to manipulate the internal structure of the picture, nor is it possible to display or manipulate parts of the picture. An example of the use of BLOBs is given in Figure 18.12.

Limited operations

The relational model has only a fixed set of operations, such as set and tuple-oriented operations, operations that are provided in the SQL specification. However, SQL does not allow new operations to be specified. Again, this is too restrictive to model the behavior of many 'real world' objects. For example, a GIS application typically uses points, lines, line groups, and polygons, and needs operations for distance, intersection, and containment.

Difficulty handling recursive queries

Atomicity of data means that repeating groups are not allowed in the relational model. As a result, it is extremely difficult to handle recursive queries, that is, queries about relationships that a relation has with itself (directly or indirectly). Consider the simplified Staff relation shown in Figure 25.1(a), which stores staff numbers and the corresponding manager's staff number. How do we find all the managers who, directly or indirectly, manage staff member S005? To find the first two levels of the hierarchy, we use:

Figure 25.1
(a) Simplified Staff relation; (b) transitive closure of Staff relation.

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL

(a)

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL
S005	S003
S005	S002
S005	S001
S004	S002
S004	S001
S003	S001

(b)

```
SELECT managerStaffNo
FROM Staff
WHERE staffNo = 'S005'
UNION
SELECT managerStaffNo
FROM Staff
WHERE staffNo =
        (SELECT managerStaffNo
         FROM Staff
         WHERE staffNo = 'S005');
```

We can easily extend this approach to find the complete answer to this query. For this particular example, this approach works because we know how many levels in the hierarchy have to be processed. However, if we were to ask a more general query, such as 'For each member of staff, find all the managers who directly or indirectly manage the individual', this approach would be impossible to implement using interactive SQL. To overcome this problem, SQL can be embedded in a high-level programming language, which provides constructs to facilitate iteration (see Appendix E). Additionally, many RDBMSs provide a report writer with similar constructs. In either case, it is the application rather than the inherent capabilities of the system that provides the required functionality.

An extension to relational algebra that has been proposed to handle this type of query is the unary **transitive closure**, or **recursive closure**, operation (Merrett, 1984):

Transitive closure The transitive closure of a relation R with attributes (A_1, A_2) defined on the same domain is the relation R augmented with all tuples successively deduced by transitivity; that is, if (a, b) and (b, c) are tuples of R, the tuple (a, c) is also added to the result.

This operation cannot be performed with just a fixed number of relational algebra operations, but requires a loop along with the Join, Projection, and Union operations. The result of this operation on our simplified Staff relation is shown in Figure 25.1(b).

Impedance mismatch

In Section 5.1 we noted that SQL-92 lacked *computational completeness*. This is true with most Data Manipulation Languages (DMLs) for RDBMSs. To overcome this problem and to provide additional flexibility, the SQL standard provides embedded SQL to help develop more complex database applications (see Appendix E). However, this approach produces an **impedance mismatch** because we are mixing different programming paradigms:

- SQL is a declarative language that handles rows of data, whereas a high-level language such as 'C' is a procedural language that can handle only one row of data at a time.
- SQL and 3GLs use different models to represent data. For example, SQL provides the built-in data types Date and Interval, which are not available in traditional programming languages. Thus, it is necessary for the application program to convert between the two representations, which is inefficient both in programming effort and in the use of runtime resources. It has been estimated that as much as 30% of programming effort and code space is expended on this type of conversion (Atkinson *et al*., 1983). Furthermore, since we are using two different type systems, it is not possible to automatically type check the application as a whole.

It is argued that the solution to these problems is not to replace relational languages by row-level object-oriented languages, but to introduce set-level facilities into programming languages (Date, 2000). However, the basis of OODBMSs is to provide a much more seamless integration between the DBMS's data model and the host programming language. We return to this issue in the next chapter.

Other problems with RDBMSs

- Transactions in business processing are generally short-lived and the concurrency control primitives and protocols such as two-phase locking are not particularly suited for long-duration transactions, which are more common for complex design objects (see Section 20.4).
- Schema changes are difficult. Database administrators must intervene to change database structures and, typically, programs that access these structures must be modified to adjust to the new structures. These are slow and cumbersome processes even with current technologies. As a result, most organizations are locked into their existing database structures. Even if they are willing and able to change the way they do business to meet new requirements, they are unable to make these changes because they cannot afford the time and expense required to modify their information systems (Taylor, 1992). To meet the requirement for increased flexibility, we need a system that caters for natural schema evolution.

- RDBMSs were designed to use content-based *associative access* (that is, declarative statements with selection based on one or more predicates) and are poor at *navigational access* (that is, access based on movement between individual records). Navigational access is important for many of the complex applications we discussed in the previous section.

Of these three problems, the first two are applicable to many DBMSs, not just relational systems. In fact, there is no underlying problem with the relational model that would prevent such mechanisms being implemented.

The latest release of the SQL standard, SQL:2003, addresses some of the above deficiencies with the introduction of many new features, such as the ability to define new data types and operations as part of the data definition language, and the addition of new constructs to make the language computationally complete. We discuss SQL:2003 in detail in Section 28.4.

25.3 Object-Oriented Concepts

In this section we discuss the main concepts that occur in object-orientation. We start with a brief review of the underlying themes of abstraction, encapsulation, and information hiding.

25.3.1 Abstraction, Encapsulation, and Information Hiding

Abstraction is the process of identifying the essential aspects of an entity and ignoring the unimportant properties. In software engineering this means that we concentrate on what an object is and what it does before we decide how it should be implemented. In this way we delay implementation details for as long as possible, thereby avoiding commitments that we may find restrictive at a later stage. There are two fundamental aspects of abstraction: encapsulation and information hiding.

The concept of **encapsulation** means that an object contains both the data structure and the set of operations that can be used to manipulate it. The concept of **information hiding** means that we separate the external aspects of an object from its internal details, which are hidden from the outside world. In this way the internal details of an object can be changed without affecting the applications that use it, provided the external details remain the same. This prevents an application becoming so interdependent that a small change has enormous ripple effects. In other words information hiding provides a form of *data independence*.

These concepts simplify the construction and maintenance of applications through **modularization**. An object is a 'black box' that can be constructed and modified independently of the rest of the system, provided the external interface is not changed. In some systems, for example Smalltalk, the ideas of encapsulation and information hiding are brought together. In Smalltalk the object structure is always hidden and only the operation interface can ever be visible. In this way the object structure can be changed without affecting any applications that use the object.

There are two views of encapsulation: the object-oriented programming language (OOPL) view and the database adaptation of that view. In some OOPLs encapsulation is achieved through **Abstract Data Types** (ADTs). In this view an object has an interface part and an implementation part. The interface provides a specification of the operations that can be performed on the object; the implementation part consists of the data structure for the ADT and the functions that realize the interface. Only the interface part is visible to other objects or users. In the database view, proper encapsulation is achieved by ensuring that programmers have access only to the interface part. In this way encapsulation provides a form of *logical data independence*: we can change the internal implementation of an ADT without changing any of the applications using that ADT (Atkinson *et al.*, 1989).

Objects and Attributes 25.3.2

Many of the important object-oriented concepts stem from the Simula programming language developed in Norway in the mid-1960s to support simulation of 'real world' processes (Dahl and Nygaard, 1966), although object-oriented programming did not emerge as a new programming paradigm until the development of the Smalltalk language (Goldberg and Robson, 1983). Modules in Simula are not based on procedures as they are in conventional programming languages, but on the physical objects being modeled in the simulation. This seemed a sensible approach as the objects are the key to the simulation: each object has to maintain some information about its current **state**, and additionally has actions (**behavior**) that have to be modeled. From Simula, we have the definition of an object.

Object A uniquely identifiable entity that contains both the attributes that describe the state of a 'real world' object and the actions that are associated with it.

In the *DreamHome* case study, a branch office, a member of staff, and a property are examples of objects that we wish to model. The concept of an object is simple but, at the same time, very powerful: each object can be defined and maintained independently of the others. This definition of an object is very similar to the definition of an entity given in Section 11.1.1. However, an object encapsulates both state and behavior; an entity models only state.

The current state of an object is described by one or more **attributes** (**instance variables**). For example, the branch office at 163 Main St may have the attributes shown in Table 25.2. Attributes can be classified as simple or complex. A **simple attribute** can be a primitive type such as integer, string, real, and so on, which takes on literal values; for example, branchNo in Table 25.2 is a simple attribute with the literal value 'B003'. A **complex attribute** can contain collections and/or references. For example, the attribute SalesStaff is a **collection** of Staff objects. A **reference attribute** represents a relationship between objects and contains a value, or collection of values, which are themselves objects (for example, SalesStaff is, more precisely, a collection of references to Staff objects). A reference attribute is conceptually similar to a foreign key in the relational data model or a pointer in a programming language. An object that contains one or more complex attributes is called a **complex object** (see Section 25.3.9).

Table 25.2 Object attributes for branch instance.

Attribute	Value
branchNo	B003
street	163 Main St
city	Glasgow
postcode	G11 9QX
SalesStaff	Ann Beech; David Ford
Manager	Susan Brand

Attributes are generally referenced using the 'dot' notation. For example, the street attribute of a branch object is referenced as:

branchObject.street

25.3.3 Object Identity

A key part of the definition of an object is unique identity. In an object-oriented system, each object is assigned an **Object Identifier** (OID) when it is created that is:

- system-generated;
- unique to that object;
- invariant, in the sense that it cannot be altered during its lifetime. Once the object is created, this OID will not be reused for any other object, even after the object has been deleted;
- independent of the values of its attributes (that is, its state). Two objects could have the same state but would have different identities;
- invisible to the user (ideally).

Thus, object identity ensures that an object can always be uniquely identified, thereby automatically providing entity integrity (see Section 3.3.2). In fact, as object identity ensures uniqueness system-wide, it provides a stronger constraint than the relational data model's entity integrity, which requires only uniqueness within a relation. In addition, objects can contain, or refer to, other objects using object identity. However, for each referenced OID in the system there should always be an object present that corresponds to the OID, that is, there should be no **dangling references**. For example, in the *DreamHome* case study, we have the relationship Branch *Has* Staff. If we embed each branch object in the related staff object, then we encounter the problems of information redundancy and update anomalies discussed in Section 13.2. However, if we instead embed the OID of the branch object in the related staff object, then there continues to be only one instance of each branch object in the system and consistency can be maintained more easily. In this way, objects can be *shared* and OIDs can be used to maintain referential integrity (see Section 3.3.3). We discuss referential integrity in OODBMSs in Section 25.6.2.

There are several ways in which object identity can be implemented. In an RDBMS, object identity is *value-based*: the primary key is used to provide uniqueness of each tuple in a relation. Primary keys do not provide the type of object identity that is required in object-oriented systems. First, as already noted, the primary key is only unique within a relation, not across the entire system. Second, the primary key is generally chosen from the attributes of the relation, making it dependent on object state. If a potential key is subject to change, identity has to be simulated by unique identifiers, such as the branch number branchNo, but as these are not under system control there is no guarantee of protection against violations of identity. Furthermore, simulated keys such as B001, B002, B003, have little semantic meaning to the user.

Other techniques that are frequently used in programming languages to support identity are variable names and pointers (or virtual memory addresses), but these approaches also compromise object identity (Khoshafian and Abnous, 1990). For example, in 'C' and C++ an OID is a physical address in the process memory space. For most database purposes this address space is too small: scalability requires that OIDs be valid across storage volumes, possibly across different computers for distributed DBMSs. Further, when an object is deleted, the memory formerly occupied by it should be reused, and so a new object may be created and allocated to the same space as the deleted object occupied. All references to the old object, which became invalid after the deletion, now become valid again, but unfortunately referencing the wrong object. In a similar way moving an object from one address to another invalidates the object's identity. What is required is a *logical object identifier* that is independent of both state and location. We discuss logical and physical OIDs in Section 26.2.

There are several advantages to using OIDs as the mechanism for object identity:

- *They are efficient* OIDs require minimal storage within a complex object. Typically, they are smaller than textual names, foreign keys, or other semantic-based references.
- *They are fast* OIDs point to an actual address or to a location within a table that gives the address of the referenced object. This means that objects can be located quickly whether they are currently stored in local memory or on disk.
- *They cannot be modified by the user* If the OIDs are system-generated and kept invisible, or at least read-only, the system can ensure entity and referential integrity more easily. Further, this avoids the user having to maintain integrity.
- *They are independent of content* OIDs do not depend upon the data contained in the object in any way. This allows the value of every attribute of an object to change, but for the object to remain the same object with the same OID.

Note the potential for ambiguity that can arise from this last property: two objects can appear to be the same to the user (all attribute values are the same), yet have different OIDs and so be different objects. If the OIDs are invisible, how does the user distinguish between these two objects? From this we may conclude that primary keys are still required to allow users to distinguish objects. With this approach to designating an object, we can distinguish between object identity (sometimes called object equivalence) and object equality. Two objects are **identical** (equivalent) if and only if they are the same object (denoted by '='), that is their OIDs are the same. Two objects are **equal** if their states are the same (denoted by '= ='). We can also distinguish between shallow and deep equality:

objects have **shallow equality** if their states contain the same values when we exclude references to other objects; objects have **deep equality** if their states contain the same values and if related objects also contain the same values.

25.3.4 Methods and Messages

An object encapsulates both data and functions into a self-contained package. In object technology, functions are usually called **methods**. Figure 25.2 provides a conceptual representation of an object, with the attributes on the inside protected from the outside by the methods. Methods define the **behavior** of the object. They can be used to change the object's state by modifying its attribute values, or to query the values of selected attributes. For example, we may have methods to add a new property for rent at a branch, to update a member of staff's salary, or to print out a member of staff's details.

A method consists of a name and a body that performs the behavior associated with the method name. In an object-oriented language, the body consists of a block of code that carries out the required functionality. For example, Figure 25.3 represents the method to update a member of staff's salary. The name of the method is updateSalary, with an input parameter *increment*, which is added to the **instance variable** salary to produce a new salary.

Messages are the means by which objects communicate. A message is simply a request from one object (the sender) to another object (the receiver) asking the second object to execute one of its methods. The sender and receiver may be the same object. Again, the dot notation is generally used to access a method. For example, to execute the updateSalary

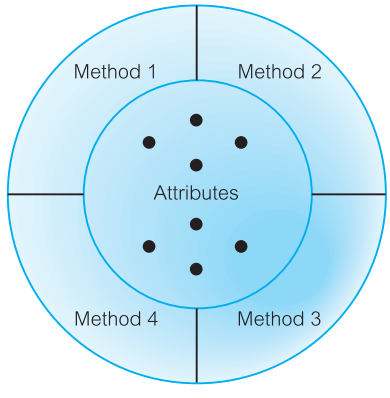


Figure 25.2
Object showing attributes and methods.

```
method void updateSalary(float increment)
{
        salary = salary + increment;
}
```

Figure 25.3
Example of a method.

method on a Staff object, staffObject, and pass the method an increment value of 1000, we write:

staffObject.updateSalary(1000)

In a traditional programming language, a message would be written as a function call:

updateSalary(staffObject, 1000)

25.3.5 Classes

In Simula, classes are blueprints for defining a set of similar objects. Thus, objects that have the same attributes and respond to the same messages can be grouped together to form a **class**. The attributes and associated methods are defined once for the class rather than separately for each object. For example, all branch objects would be described by a single Branch class. The objects in a class are called **instances** of the class. Each instance has its own value(s) for each attribute, but shares the same attribute names and methods with other instances of the class, as illustrated in Figure 25.4.

In the literature, the terms 'class' and 'type' are often used synonymously, although some authors make a distinction between the two terms as we now describe. A *type*

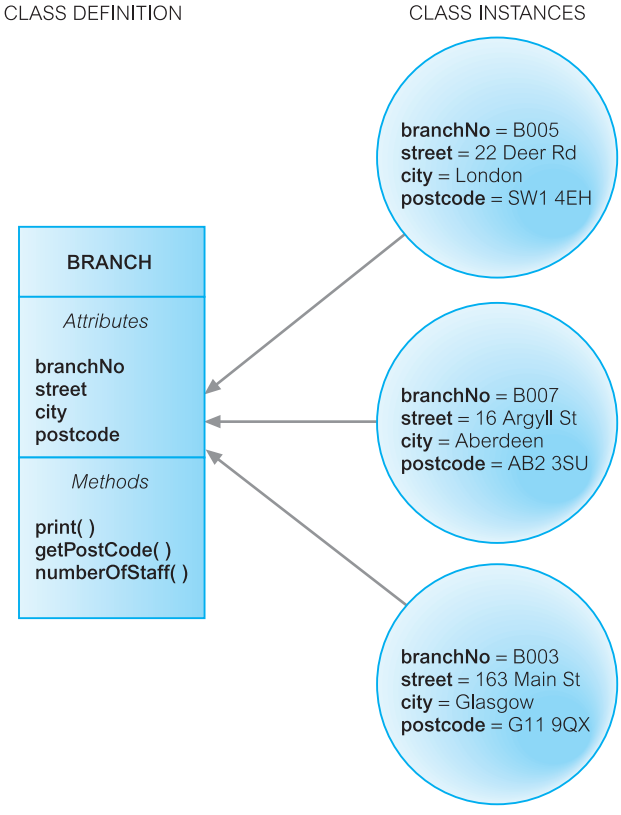


Figure 25.4 Class instances share attributes and methods.

corresponds to the notion of an abstract data type (Atkinson and Buneman, 1989). In programming languages, a variable is declared to be of a particular type. The compiler can use this type to check that the operations performed on the variable are compatible with its type, thus helping to ensure the correctness of the software. On the other hand, a *class* is a blueprint for creating objects and provides methods that can be applied on the objects. Thus, a class is referred to at runtime rather than compile time.

In some object-oriented systems, a class is also an object and has its own attributes and methods, referred to as **class attributes** and **class methods**, respectively. Class attributes describe the general characteristics of the class, such as totals or averages; for example, in the class Branch we may have a class attribute for the total number of branches. Class methods are used to change or query the state of class attributes. There are also special class methods to create new instances of the class and to destroy those that are no longer required. In an object-oriented language, a new instance is normally created by a method called new. Such methods are usually called **constructors**. Methods for destroying objects and reclaiming the space occupied are typically called **destructors**. Messages sent to a class method are sent to the class rather than an instance of a class, which implies that the class is an instance of a higher-level class, called a **metaclass**.

25.3.6 Subclasses, Superclasses, and Inheritance

Some objects may have similar but not identical attributes and methods. If there is a large degree of similarity, it would be useful to be able to share the common properties (attributes and methods). **Inheritance** allows one class to be defined as a special case of a more general class. These special cases are known as **subclasses** and the more general cases are known as **superclasses**. The process of forming a superclass is referred to as **generalization** and the process of forming a subclass is **specialization**. By default, a subclass inherits all the properties of its superclass(es) and, additionally, defines its own unique properties. However, as we see shortly, a subclass can also redefine inherited properties. All instances of the subclass are also instances of the superclass. Further, the *principle of substitutability* states that we can use an instance of the subclass whenever a method or a construct expects an instance of the superclass.

The concepts of superclass, subclass, and inheritance are similar to those discussed for the Enhanced Entity–Relationship (EER) model in Chapter 12, except that in the object-oriented paradigm inheritance covers both state and behavior. The relationship between the subclass and superclass is sometimes referred to as **A KIND OF** (AKO) relationship, for example a Manager is AKO Staff. The relationship between an instance and its class is sometimes referred to as **IS-A**; for example, Susan Brand IS-A Manager.

There are several forms of inheritance: single inheritance, multiple inheritance, repeated inheritance, and selective inheritance. Figure 25.5 shows an example of **single inheritance**, where the subclasses Manager and SalesStaff inherit the properties of the superclass Staff. The term ‘single inheritance’ refers to the fact that the subclasses inherit from no more than one superclass. The superclass Staff could itself be a subclass of a superclass, Person, thus forming a **class hierarchy**.

Figure 25.6 shows an example of **multiple inheritance** where the subclass SalesManager inherits properties from both the superclasses Manager and SalesStaff. The provision of a

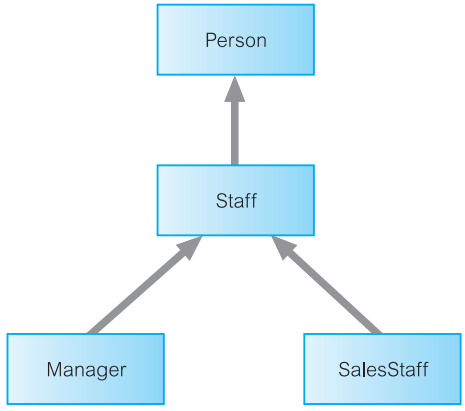


Figure 25.5
Single inheritance.

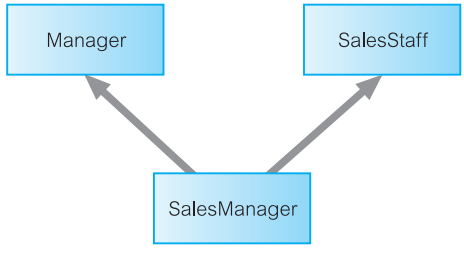


Figure 25.6
Multiple inheritance.

mechanism for multiple inheritance can be quite problematic as it has to provide a way of dealing with conflicts that arise when the superclasses contain the same attributes or methods. Not all object-oriented languages and DBMSs support multiple inheritance as a matter of principle. Some authors claim that multiple inheritance introduces a level of complexity that is hard to manage safely and consistently. Others argue that it is required to model the 'real world', as in this example. Those languages that do support it, handle conflict in a variety of ways, such as:

- Include both attribute/method names and use the name of the superclass as a qualifier. For example, if bonus is an attribute of both Manager and SalesStaff, the subclass SalesManager could inherit bonus from both superclasses and qualify the instance of bonus in SalesManager as either Manager.bonus or SalesStaff.bonus.
- Linearize the inheritance hierarchy and use single inheritance to avoid conflicts. With this approach, the inheritance hierarchy of Figure 25.6 would be interpreted as:

  SalesManager → Manager → SalesStaff

  or

  SalesManager → SalesStaff → Manager

  With the previous example, SalesManager would inherit one instance of the attribute bonus, which would be from Manager in the first case, and SalesStaff in the second case.

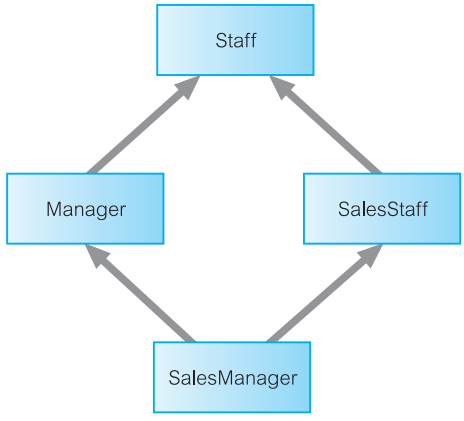


Figure 25.7
Repeated inheritance.

- Require the user to redefine the conflicting attributes or methods.
- Raise an error and prohibit the definition until the conflict is resolved.

Repeated inheritance is a special case of multiple inheritance where the superclasses inherit from a common superclass. Extending the previous example, the classes Manager and SalesStaff may both inherit properties from a common superclass Staff, as illustrated in Figure 25.7. In this case, the inheritance mechanism must ensure that the SalesManager class does not inherit properties from the Staff class twice. Conflicts can be handled as discussed for multiple inheritance.

Selective inheritance allows a subclass to inherit a limited number of properties from the superclass. This feature may provide similar functionality to the view mechanism discussed in Section 6.4 by restricting access to some details but not others.

25.3.7 Overriding and Overloading

As we have just mentioned, properties (attributes and methods) are automatically inherited by subclasses from their superclasses. However, it is possible to redefine a property in the subclass. In this case, the definition of the property in the subclass is the one used. This process is called **overriding**. For example, we might define a method in the Staff class to increment salary based on a commission:

```
method void giveCommission(float branchProfit) {
    salary = salary + 0.02 * branchProfit;
}
```

However, we may wish to perform a different calculation for commission in the Manager subclass. We can do this by redefining, or *overriding*, the method giveCommission in the Manager subclass:

```
method void giveCommission(float branchProfit) {
    salary = salary + 0.05 * branchProfit;
}
```

```
method void print( )  {
    printf("Branch number: %s\n", branchNo);
    printf("Street: %s\n", street);
    printf("City: %s\n", city);
    printf("Postcode: %s\n", postcode);
}
```

(a)

```
method void print( )  {
    printf("Staff number: %s\n", staffNo);
    printf("First name: %s\n", fName);
    printf("Last name: %s\n", lName);
    printf("Position: %s\n", position);
    printf("Sex: %c\n", sex);
    printf("Date of birth: %s\n", DOB);
    printf("Salary: %f\n", salary);
}
```

(b)

Figure 25.8 Overloading print method: (a) for Branch object; (b) for Staff object.

The ability to factor out common properties of several classes and form them into a superclass that can be shared with subclasses can greatly reduce redundancy within systems and is regarded as one of the main advantages of object-orientation. Overriding is an important feature of inheritance as it allows special cases to be handled easily with minimal impact on the rest of the system.

Overriding is a special case of the more general concept of **overloading**. Overloading allows the name of a method to be reused within a class definition or across class definitions. This means that a single message can perform different functions depending on which object receives it and, if appropriate, what parameters are passed to the method. For example, many classes will have a print method to print out the relevant details for an object, as shown in Figure 25.8.

Overloading can greatly simplify applications, since it allows the same name to be used for the same operation irrespective of what class it appears in, thereby allowing context to determine which meaning is appropriate at any given moment. This saves having to provide unique names for methods such as printBranchDetails or printStaffDetails for what is in essence the same functional operation.

Polymorphism and Dynamic Binding 25.3.8

Overloading is a special case of the more general concept of **polymorphism**, from the Greek meaning 'having many forms'. There are three types of polymorphism: operation, inclusion, and parametric (Cardelli and Wegner, 1985). Overloading, as in the previous example, is a type of **operation** (or ***ad hoc***) **polymorphism**. A method defined in a superclass and inherited in its subclasses is an example of **inclusion polymorphism**. **Parametric polymorphism**, or **genericity** as it is sometimes called, uses types as parameters in generic type, or class, declarations. For example, the following template definition:

```
template <type T>
T max(x:T, y:T) {
   if (x > y)      return x;
   else            return y;
}
```

defines a generic function max that takes two parameters of type T and returns the maximum of the two values. This piece of code does not actually establish any methods. Rather, the generic description acts as a template for the later establishment of one or more different methods of different types. Actual methods are instantiated as:

```
int max(int, int);          // instantiate max function for two integer types
real max(real, real);       // instantiate max function for two real types
```

The process of selecting the appropriate method based on an object's type is called **binding**. If the determination of an object's type can be deferred until runtime (rather than compile time), the selection is called **dynamic (late) binding**. For example, consider the class hierarchy of Staff with subclasses Manager and SalesStaff shown in Figure 25.5, and assume that each class has its own print method to print out relevant details. Further assume that we have a list consisting of an arbitrary number of objects, *n* say, from this hierarchy. In a conventional programming language, we would need a CASE statement or a nested IF statement to print out the corresponding details:

```
FOR i = 1 TO n DO
SWITCH (list[i]. type) {
    CASE staff:         printStaffDetails(list[i].object); break;
    CASE manager:       printManagerDetails(list[i].object); break;
    CASE salesPerson:   printSalesStaffDetails(list[i].object); break;
}
```

If a new type is added to the list, we have to extend the CASE statement to handle the new type, forcing recompilation of this piece of software. If the language supports dynamic binding and overloading, we can overload the print methods with the single name print and replace the CASE statement with the line:

```
list[i].print()
```

Furthermore, with this approach we can add any number of new types to the list and, provided we continue to overload the print method, no recompilation of this code is required. Thus, the concept of polymorphism is orthogonal to (that is, independent of) inheritance.

25.3.9 Complex Objects

There are many situations where an object consists of subobjects or components. A complex object is an item that is viewed as a single object in the 'real world' but combines with other objects in a set of complex **A-PART-OF** relationships (APO). The objects contained may themselves be complex objects, resulting in an **A-PART-OF hierarchy**. In an object-oriented system, a contained object can be handled in one of two ways. First, it can be encapsulated within the complex object and thus form part of the complex object. In this case, the structure of the contained object is part of the structure of the complex object and can be accessed only by the complex object's methods. On the other hand, a contained object can be considered to have an independent existence from the complex object. In this

case, the object is not stored directly in the parent object but only its OID. This is known as **referential sharing** (Khoshafian and Valduriez, 1987). The contained object has its own structure and methods, and can be owned by several parent objects.

These types of complex object are sometimes referred to as **structured complex** objects, since the system knows the composition. The term **unstructured complex object** is used to refer to a complex object whose structure can be interpreted only by the application program. In the database context, unstructured complex objects are sometimes known as Binary Large Objects (BLOBs), which we discussed in Section 25.2.

25.4 Storing Objects in a Relational Database

One approach to achieving persistence with an object-oriented programming language, such as C++ or Java, is to use an RDBMS as the underlying storage engine. This requires mapping class instances (that is, objects) to one or more tuples distributed over one or more relations. This can be problematic as we discuss in this section. For the purposes of discussion, consider the inheritance hierarchy shown in Figure 25.9, which has a Staff superclass and three subclasses: Manager, SalesPersonnel, and Secretary.

To handle this type of class hierarchy, we have two basics tasks to perform:

- Design the relations to represent the class hierarchy.
- Design how objects will be accessed, which means:
  - writing code to decompose the objects into tuples and store the decomposed objects in relations;
  - writing code to read tuples from the relations and reconstruct the objects.

We now describe these two tasks in more detail.

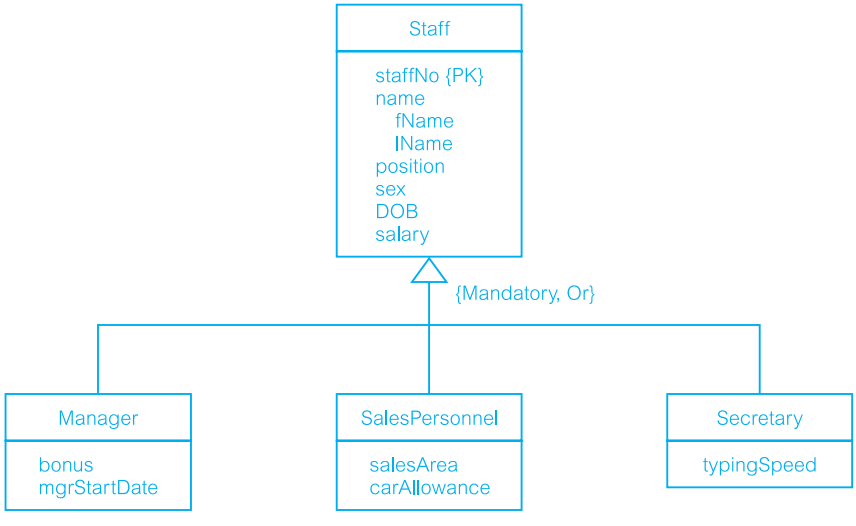


Figure 25.9 Sample inheritance hierarchy for Staff.

25.4.1 Mapping Classes to Relations

There are a number of strategies for mapping classes to relations, although each results in a loss of semantic information. The code to make objects persistent and to read the objects back from the database is dependent on the strategy chosen. We consider three alternatives:

(1) Map each class or subclass to a relation.

(2) Map each subclass to a relation.

(3) Map the hierarchy to a single relation.

Map each class or subclass to a relation

One approach is to map each class or subclass to a relation. For the hierarchy given in Figure 25.9, this would give the following four relations (with the primary key underlined):

Staff (staffNo, fName, lName, position, sex, DOB, salary)
Manager (staffNo, bonus, mgrStartDate)
SalesPersonnel (staffNo, salesArea, carAllowance)
Secretary (staffNo, typingSpeed)

We assume that the underlying data type of each attribute is supported by the RDBMS, although this may not be the case – in which case we would need to write additional code to handle the transformation of one data type to another.

Unfortunately, with this relational schema we have lost semantic information: it is no longer clear which relation represents the superclass and which relations represent the subclasses. We would therefore have to build this knowledge into each application, which as we have said on other occasions can lead to duplication of code and potential for inconsistencies to arise.

Map each subclass to a relation

A second approach is to map each subclass to a relation. For the hierarchy given in Figure 25.9, this would give the following three relations:

Manager (staffNo, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate)
SalesPersonnel (staffNo, fName, lName, position, sex, DOB, salary, salesArea, carAllowance)
Secretary (staffNo, fName, lName, position, sex, DOB, salary, typingSpeed)

Again, we have lost semantic information in this mapping: it is no longer clear that these relations are subclasses of a single generic class. In this case, to produce a list of all staff we would have to select the tuples from each relation and then union the results together.

Map the hierarchy to a single relation

A third approach is to map the entire inheritance hierarchy to a single relation, giving in this case:

Staff (staffNo, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate, salesArea, carAllowance, typingSpeed, typeFlag)

The attribute typeFlag is a discriminator to distinguish which type each tuple is (for example, it may contain the value 1 for a Manager tuple, 2 for a SalesPersonnel tuple, and 3 for a Secretary tuple). Again, we have lost semantic information in this mapping. Further, this mapping will produce an unwanted number of nulls for attributes that do not apply to that tuple. For example, for a Manager tuple, the attributes salesArea, carAllowance, and typingSpeed will be null.

Accessing Objects in the Relational Database 25.4.2

Having designed the structure of the relational database, we now need to insert objects into the database and then provide a mechanism to read, update, and delete the objects. For example, to insert an object into the first relational schema in the previous section (that is, where we have created a relation for each class), the code may look something like the following using programmatic SQL (see Appendix E):

```
Manager* pManager = new Manager;        // create a new Manager object
. . . code to set up the object . . .
EXEC SQL INSERT INTO Staff VALUES (:pManager->staffNo, :pManager->fName,
   :pManager->lName, :pManager->position, :pManager->sex, :pManager->DOB,
   :pManager->salary);
EXEC SQL INSERT INTO Manager VALUES (:pManager->bonus,
   :pManager->mgrStartDate);
```

On the other hand, if Manager had been declared as a persistent class then the following (indicative) statement would make the object persistent in an OODBMS:

```
Manager* pManager = new Manager;
```

In Section 26.3, we examine different approaches for declaring persistent classes. If we now wished to retrieve some data from the relational database, say the details for managers with a bonus in excess of £1000, the code may look something like the following:

```
Manager* pManager = new Manager;                    // create a new Manager object
EXEC SQL WHENEVER NOT FOUND GOTO done;   // set up error handling
EXEC SQL DECLARE managerCursor                      // create cursor for SELECT
   CURSOR FOR
      SELECT staffNo, fName, lName, salary, bonus
      FROM Staff s, Manager m                       // Need to join Staff and Manager
      WHERE s.staffNo = m.staffNo AND bonus > 1000;
EXEC SQL OPEN managerCursor;
for ( ; ; ) {
      EXEC SQL FETCH managerCursor     // fetch the next record in the result
      INTO :staffNo, :fName, :lName, :salary, :bonus;
      pManager->staffNo = :staffNo;             // transfer the data to the Manager object
```

```
        pManager->fName = :fName;
        pManager->lName = :lName;
        pManager->salary = :salary;
        pManager->bonus = :bonus;
        strcpy(pManager->position, "Manager");
    }
    EXEC SQL CLOSE managerCursor;          // close the cursor before completing
```

On the other hand, to retrieve the same set of data in an OODBMS, we may write the following code:

```
os_Set<Manager*> &highBonus
  = managerExtent->query("Manager*", "bonus > 1000", db1);
```

This statement queries the extent of the Manager class (managerExtent) to find the required instances (bonus > 1000) from the database (in this example, db1). The commercial OODBMS ObjectStore has a collection template class os_Set, which has been instantiated in this example to contain pointers to Manager objects <Manager*>. In Section 27.3 we provide additional details of object persistence and object retrieval with ObjectStore.

The above examples have been given to illustrate the complexities involved in mapping an object-oriented language to a relational database. The OODBMS approach that we discuss in the next two chapters attempts to provide a more seamless integration of the programming language data model and the database data model thereby removing the need for complex transformations, which, as we discussed earlier, could account for as much as 30% of programming effort.

25.5 Next-Generation Database Systems

In the late 1960s and early 1970s, there were two mainstream approaches to constructing DBMSs. The first approach was based on the hierarchical data model, typified by IMS (Information Management System) from IBM, in response to the enormous information storage requirements generated by the Apollo space program. The second approach was based on the network data model, which attempted to create a database standard and resolve some of the difficulties of the hierarchical model, such as its inability to represent complex relationships effectively. Together, these approaches represented the **first generation** of DBMSs. However, these two models had some fundamental disadvantages:

- complex programs had to be written to answer even simple queries based on navigational record-oriented access;
- there was minimal data independence;
- there was no widely accepted theoretical foundation.

In 1970, Codd produced his seminal paper on the relational data model. This paper was very timely and addressed the disadvantages of the former approaches, in particular their lack of data independence. Many experimental relational DBMSs were implemented thereafter, with the first commercial products appearing in the late 1970s and early 1980s.

Now there are over a hundred relational DBMSs for both mainframe and PC environments, though some are stretching the definition of the relational model. Relational DBMSs are referred to as **second-generation** DBMSs.

However, as we discussed in Section 25.2, RDBMSs have their failings, particularly their limited modeling capabilities. There has been much research attempting to address this problem. In 1976, Chen presented the Entity–Relationship model that is now a widely accepted technique for database design, and the basis for the methodology presented in Chapters 15 and 16 of this book (Chen, 1976). In 1979, Codd himself attempted to address some of the failings in his original work with an extended version of the relational model called RM/T (Codd, 1979), and thereafter RM/V2 (Codd, 1990). The attempts to provide a data model that represents the 'real world' more closely have been loosely classified as **semantic data modeling**. Some of the more famous models are:

- the Semantic Data Model (Hammer and McLeod, 1981);
- the Functional Data Model (Shipman, 1981), which we examine in Section 26.1.2;
- the Semantic Association Model (Su, 1983).

In response to the increasing complexity of database applications, two 'new' data models have emerged: the **Object-Oriented Data Model** (OODM) and the **Object-Relational Data Model** (ORDM), previously referred to as the **Extended Relational Data Model** (ERDM). However, unlike previous models, the actual composition of these models is not clear. This evolution represents **third-generation** DBMSs, as illustrated in Figure 25.10.

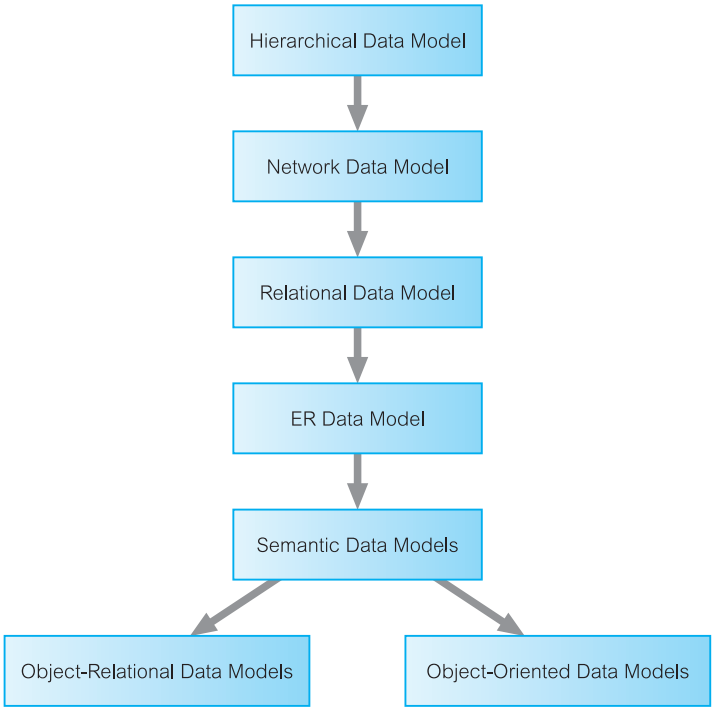


Figure 25.10
History of data models.

There is currently considerable debate between the OODBMS proponents and the relational supporters, which resembles the network/relational debate of the 1970s. Both sides agree that traditional RDBMSs are inadequate for certain types of application. However, the two sides differ on the best solution. The OODBMS proponents claim that RDBMSs are satisfactory for standard business applications but lack the capability to support more complex applications. The relational supporters claim that relational technology is a necessary part of any real DBMS and that complex applications can be handled by extensions to the relational model.

At present, relational/object-relational DBMSs form the dominant system and object-oriented DBMSs have their own particular niche in the marketplace. If OODBMSs are to become dominant they must change their image from being systems solely for complex applications to being systems that can also accommodate standard business applications with the same tools and the same ease of use as their relational counterparts. In particular, they must support a declarative query language compatible with SQL. We devote Chapters 26 and 27 to a discussion of OODBMSs and Chapter 28 to ORDBMSs.

25.6 Object-Oriented Database Design

In this section we discuss how to adapt the methodology presented in Chapters 15 and 16 for an OODBMS. We start the discussion with a comparison of the basis for our methodology, the Enhanced Entity–Relationship model, and the main object-oriented concepts. In Section 25.6.2 we examine the relationships that can exist between objects and how referential integrity can be handled. We conclude this section with some guidelines for identifying methods.

25.6.1 Comparison of Object-Oriented Data Modeling and Conceptual Data Modeling

The methodology for conceptual and logical database design presented in Chapters 15 and 16, which was based on the Enhanced Entity–Relationship (EER) model, has similarities with Object-Oriented Data Modeling (OODM). Table 25.3 compares OODM with Conceptual Data Modeling (CDM). The main difference is the encapsulation of both state and behavior in an object, whereas CDM captures only state and has no knowledge of behavior. Thus, CDM has no concept of messages and consequently no provision for encapsulation.

The similarity between the two approaches makes the conceptual and logical data modeling methodology presented in Chapters 15 and 16 a reasonable basis for a methodology for object-oriented database design. Although this methodology is aimed primarily at relational database design, the model can be mapped with relative simplicity to the network and hierarchical models. The logical data model produced had many-to-many relationships and recursive relationships removed (Step 2.1). These are unnecessary changes for object-oriented modeling and can be omitted, as they were introduced because

Table 25.3 Comparison of OODM and CDM.

OODM	CDM	Difference
Object	Entity	Object includes behavior
Attribute	Attribute	None
Association	Relationship	Associations are the same but inheritance in OODM includes both state and behavior
Message		No corresponding concept in CDM
Class	Entity type/Supertype	None
Instance	Entity	None
Encapsulation		No corresponding concept in CDM

of the limited modeling power of the traditional data models. The use of normalization in the methodology is still important and should not be omitted for object-oriented database design. Normalization is used to improve the model so that it satisfies various constraints that avoid unnecessary duplication of data. The fact that we are dealing with objects does not mean that redundancy is acceptable. In object-oriented terms, second and third normal form should be interpreted as:

> 'Every attribute in an object is dependent on the object identity.'

Object-oriented database design requires the database schema to include both a description of the object data structure and constraints, and the object behavior. We discuss behavior modeling in Section 25.6.3.

Relationships and Referential Integrity 25.6.2

Relationships are represented in an object-oriented data model using **reference attributes** (see Section 25.3.2), typically implemented using OIDs. In the methodology presented in Chapters 15 and 16, we decomposed all non-binary relationships (for example, ternary relationships) into binary relationships. In this section we discuss how to represent binary relationships based on their cardinality: one-to-one (1:1), one-to-many (1:*), and many-to-many (*:*).

1:1 relationships

A 1:1 relationship between objects A and B is represented by adding a reference attribute to object A and, to maintain referential integrity, a reference attribute to object B. For example, there is a 1:1 relationship between Manager and Branch, as represented in Figure 25.11.

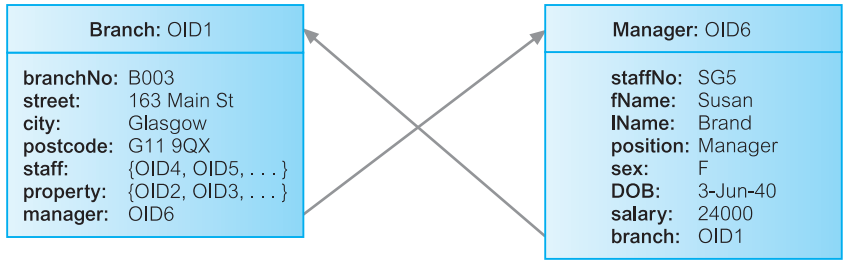


Figure 25.11
A 1:1 relationship between Manager and Branch.

1:* relationships

A 1:* relationship between objects A and B is represented by adding a reference attribute to object B and an attribute containing a set of references to object A. For example, there are 1:* relationships represented in Figure 25.12, one between Branch and SalesStaff, and the other between SalesStaff and PropertyForRent.

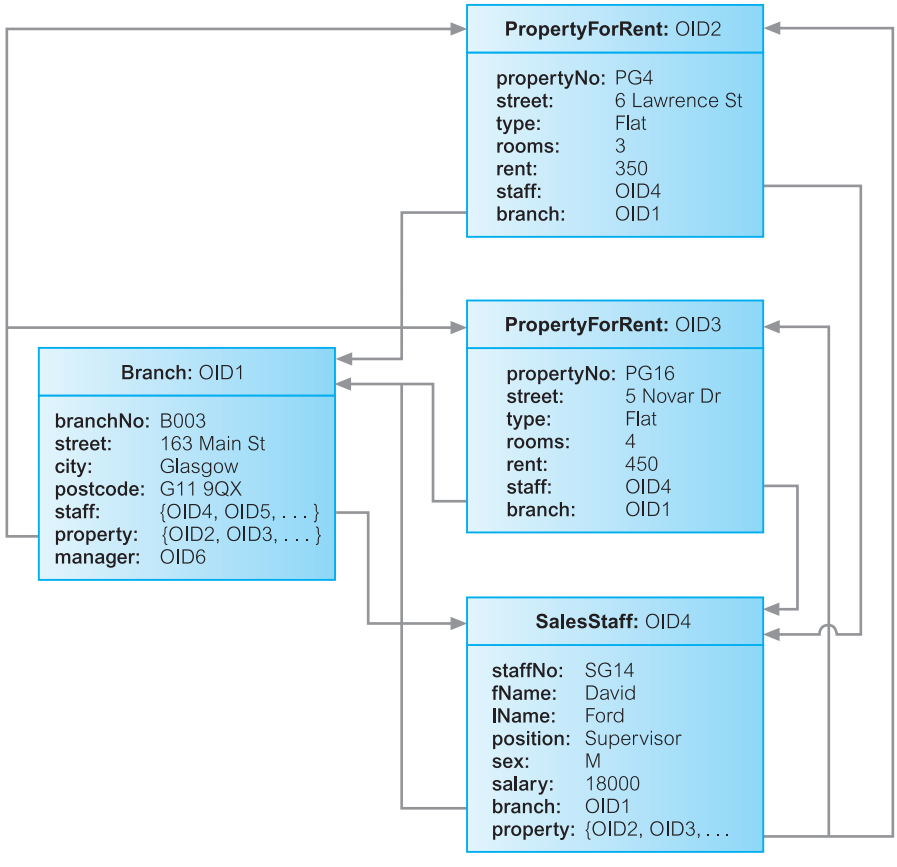


Figure 25.12
1:* relationships between Branch, SalesStaff, and PropertyForRent.

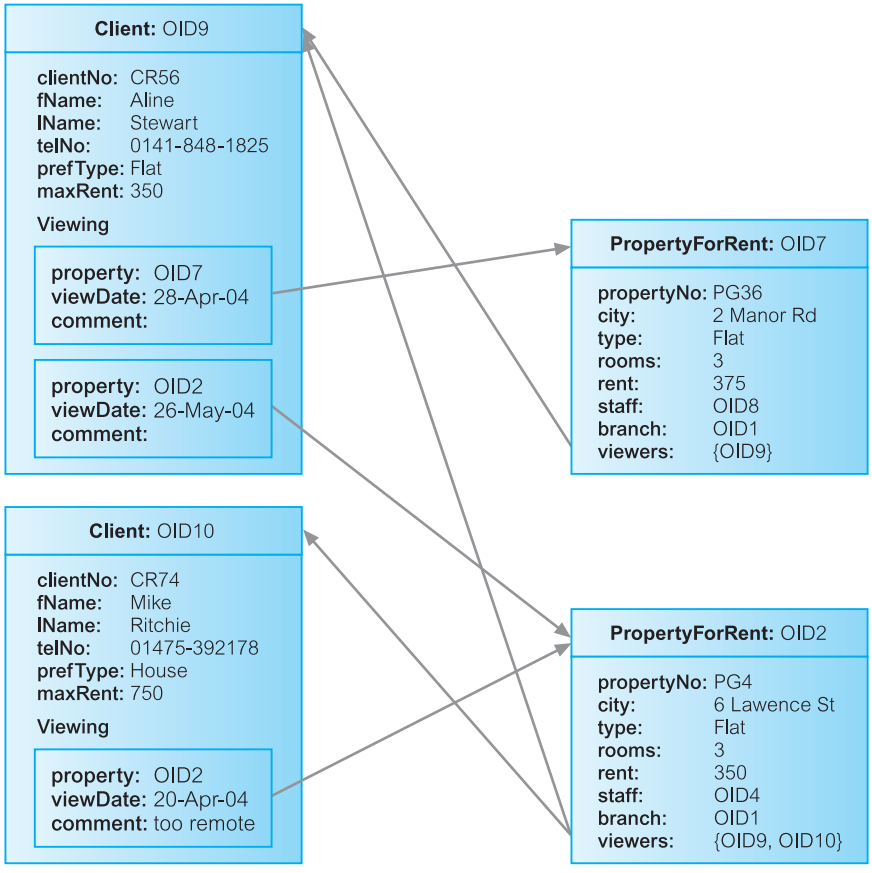


Figure 25.13
A *:* relationship between Client and PropertyForRent.

: relationships

A *:* relationship between objects A and B is represented by adding an attribute containing a set of references to each object. For example, there is a *:* relationship between Client and PropertyForRent, as represented in Figure 25.13. For relational database design, we would decompose the *:* relationship into two 1:* relationships linked by an intermediate entity. It is also possible to represent this model in an OODBMS, as shown in Figure 25.14.

Referential integrity

In Section 3.3.3 we discussed referential integrity in terms of primary and foreign keys. Referential integrity requires that any referenced object must exist. For example, consider the 1:1 relationship between Manager and Branch in Figure 25.11. The Branch instance, OID1, references a Manager instance, OID6. If the user deletes this Manager instance without updating the Branch instance accordingly, referential integrity is lost. There are several techniques that can be used to handle referential integrity:

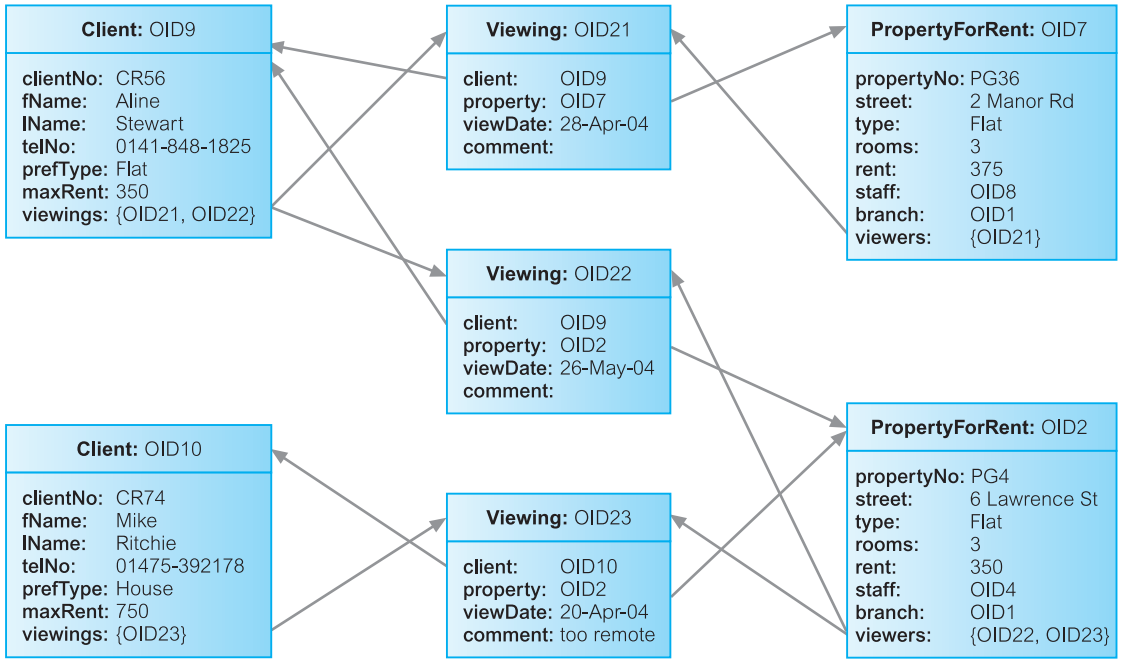


Figure 25.14
Alternative design of *:* relationship with intermediate class.

- *Do not allow the user to explicitly delete objects* In this case the system is responsible for 'garbage collection'; in other words, the system automatically deletes objects when they are no longer accessible by the user. This is the approach taken by GemStone.
- *Allow the user to delete objects when they are no longer required* In this case the system may detect an invalid reference automatically and set the reference to NULL (the null pointer) or disallow the deletion. The Versant OODBMS uses this approach to enforce referential integrity.
- *Allow the user to modify and delete objects and relationships when they are no longer required* In this case the system automatically maintains the integrity of objects, possibly using inverse attributes. For example, in Figure 25.11 we have a relationship from Branch to Manager and an inverse relationship from Manager to Branch. When a Manager object is deleted, it is easy for the system to use this inverse relationship to adjust the reference in the Branch object accordingly. The Ontos, Objectivity/DB, and ObjectStore OODBMSs provide this form of integrity, as does the ODMG Object Model (see Section 27.2).

25.6.3 Behavioral Design

The EER approach by itself is insufficient to complete the design of an object-oriented database. The EER approach must be supported with a technique that identifies and documents the behavior of each class of object. This involves a detailed analysis of the

processing requirements of the enterprise. In a conventional data flow approach using Data Flow Diagrams (DFDs), for example, the processing requirements of the system are analyzed separately from the data model. In object-oriented analysis, the processing requirements are mapped on to a set of methods that are unique for each class. The methods that are visible to the user or to other objects (**public methods**) must be distinguished from methods that are purely internal to a class (**private methods**). We can identify three types of public and private method:

- constructors and destructors;
- access;
- transform.

Constructors and destructors

Constructor methods generate new instances of a class and each new instance is given a unique OID. Destructor methods delete class instances that are no longer required. In some systems, destruction is an automatic process: whenever an object becomes inaccessible from other objects, it is automatically deleted. We referred to this previously as garbage collection.

Access methods

Access methods return the value of an attribute or set of attributes of a class instance. It may return a single attribute value, multiple attribute values, or a collection of values. For example, we may have a method getSalary for a class SalesStaff that returns a member of staff's salary, or we may have a method getContactDetails for a class Person that returns a person's address and telephone number. An access method may also return data relating to the class. For example, we may have a method getAverageSalary for a class SalesStaff that calculates the average salary of all sales staff. An access method may also derive data from an attribute. For example, we may have a method getAge for Person that calculates a person's age from the date of birth. Some systems automatically generate a method to access each attribute. This is the approach taken in the SQL:2003 standard, which provides an automatic *observer* (get) method for each attribute of each new data type (see Section 28.4).

Transform methods

Transform methods change (transform) the state of a class instance. For example, we may have a method incrementSalary for the SalesStaff class that increases a member of staff's salary by a specified amount. Some systems automatically generate a method to update each attribute. Again, this is the approach taken in the SQL:2003 standard, which provides an automatic *mutator* (put) method for each attribute of each new data type (see Section 28.4).

Identifying methods

There are several methodologies for identifying methods, which typically combine the following approaches:

- identify the classes and determine the methods that may be usefully provided for each class;
- decompose the application in a top-down fashion and determine the methods that are required to provide the required functionality.

For example, in the *DreamHome* case study we identified the operations that are to be undertaken at each branch office. These operations ensure that the appropriate information is available to manage the office efficiently and effectively, and to support the services provided to owners and clients (see Appendix A). This is a top-down approach: we interviewed the relevant users and, from that, determined the operations that are required. Using the knowledge of these required operations and using the EER model, which has identified the classes that were required, we can now start to determine what methods are required and to which class each method should belong.

A more complete description of identifying methods is outside the scope of this book. There are several methodologies for object-oriented analysis and design, and the interested reader is referred to Rumbaugh *et al*. (1991), Coad and Yourdon (1991), Graham (1993), Blaha and Premerlani (1997), and Jacobson *et al*. (1999).

25.7 Object-Oriented Analysis and Design with UML

In this book we have promoted the use of the UML (Unified Modeling Language) for ER modeling and conceptual database design. As we noted at the start of Chapter 11, UML represents a unification and evolution of several object-oriented analysis and design methods that appeared in the late 1980s and early 1990s, particularly the Booch method from Grady Booch, the Object Modeling Technique (OMT) from James Rumbaugh *et al*., and Object-Oriented Software Engineering (OOSE) from Ivar Jacobson *et al*. The UML has been adopted as a standard by the Object Management Group (OMG) and has been accepted by the software community as the primary notation for modeling objects and components.

The UML is commonly defined as 'a standard language for specifying, constructing, visualizing, and documenting the artifacts of a software system'. Analogous to the use of architectural blueprints in the construction industry, the UML provides a common language for describing software models. The UML does not prescribe any particular methodology, but instead is flexible and customizable to fit any approach and it can be used in conjunction with a wide range of software lifecycles and development processes.

The primary goals in the design of the UML were to:

- Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts. For example, the UML provides *stereotypes*, which allow new elements to be defined by extending and refining the semantics of existing elements. A stereotype is enclosed in double chevrons (<< … >>).

- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the object-oriented tools market.
- Support higher-level development concepts such as collaborations, frameworks, patterns, and components.
- Integrate best practices.

In this section we briefly examine some of the components of the UML.

25.7.1 UML Diagrams

UML defines a number of diagrams, of which the main ones can be divided into the following two categories:

- *Structural diagrams*, which describe the static relationships between components. These include:
  - class diagrams,
  - object diagrams,
  - component diagrams,
  - deployment diagrams.
- *Behavioral diagrams*, which describe the dynamic relationships between components. These include:
  - use case diagrams,
  - sequence diagrams,
  - collaboration diagrams,
  - statechart diagrams,
  - activity diagrams.

We have already used the class diagram notation for ER modeling earlier in the book. In the remainder of this section we briefly discuss the remaining types of diagrams and provide examples of their use.

Object diagrams

Object diagrams model instances of classes and are used to describe the system at a particular point in time. Just as an object is an instance of a class we can view an object diagram as an instance of a class diagram. We referred to this type of diagram as a semantic net diagram in Chapter 11. Using this technique, we can validate the class diagram (ER diagram in our case) with 'real world' data and record test cases. Many object diagrams are depicted using only entities and relationships (*objects* and *associations* in the UML terminology). Figure 25.15 shows an example of an object diagram for the Staff *Manages* PropertyForRent relationship.

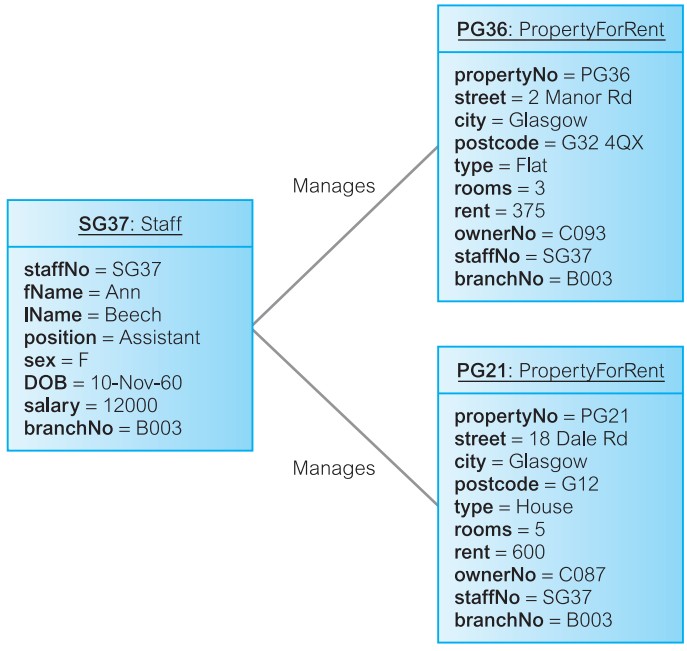


Figure 25.15
Example object diagram showing instances of the Staff *Manages* PropertyForRent relationship.

Component diagrams

Component diagrams describe the organization and dependencies among physical software components, such as source code, runtime (binary) code, and executables. For example, a component diagram can illustrate the dependency between source files and executable files, similar to the information within makefiles, which describe source code dependencies and can be used to compile and link an application. A component is represented by a rectangle with two tabs overlapping the left edge. A dependency is denoted by a dotted arrow going from a component to the component it depends on.

Deployment diagrams

Deployment diagrams depict the configuration of the runtime system, showing the hardware nodes, the components that run on these nodes, and the connections between nodes. A node is represented by a three-dimensional cube. Component and deployment diagrams can be combined as illustrated in Figure 25.16.

Use case diagrams

The UML enables and promotes (although does not mandate or even require) a use-case driven approach for modeling objects and components. Use case diagrams model the functionality provided by the system (*use cases*), the users who interact with the system (*actors*), and the association between the users and the functionality. Use cases are used in the requirements collection and analysis phase of the software development lifecycle to represent the high-level requirements of the system. More specifically, a use case specifies

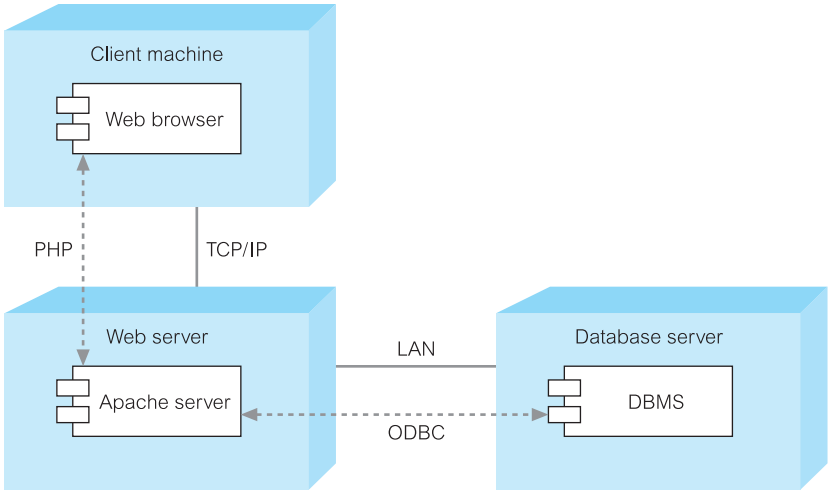


Figure 25.16 Combined component and deployment diagram.

a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor (Jacobson *et al.*, 1999).

An individual use case is represented by an ellipse, an actor by a stick figure, and an association by a line between the actor and the use case. The role of the actor is written beneath the icon. Actors are not limited to humans. If a system communicates with another application, and expects input or delivers output, then that application can also be considered an actor. A use case is typically represented by a verb followed by an object, such as View property, Lease property. An example use case diagram for Client with four use cases is shown in Figure 25.17(a) and a use case diagram for Staff in Figure 25.17(b). The use case notation is simple and therefore is a very good vehicle for communication.

Sequence diagrams

A sequence diagram models the interactions between objects over time, capturing the behavior of an individual use case. It shows the objects and the *messages* that are passed between these objects in the use case. In a sequence diagram, objects and actors are shown as columns, with vertical *lifelines* indicating the lifetime of the object over time. An activation/focus of control, which indicates when the object is performing an action, is modeled as a rectangular box on the lifeline; a lifeline is represented by a vertical dotted line extending from the object. The destruction of an object is indicated by an X at the appropriate point on its lifeline. Figure 25.18 provides an example of a sequence diagram for the Search properties use case that may have been produced during design (an earlier sequence diagram may have been produced without parameters to the messages).

Collaboration diagrams

A collaboration diagram is another type of interaction diagram, in this case showing the interactions between objects as a series of sequenced messages. This type of diagram is a cross between an object diagram and a sequence diagram. Unlike the sequence diagram,

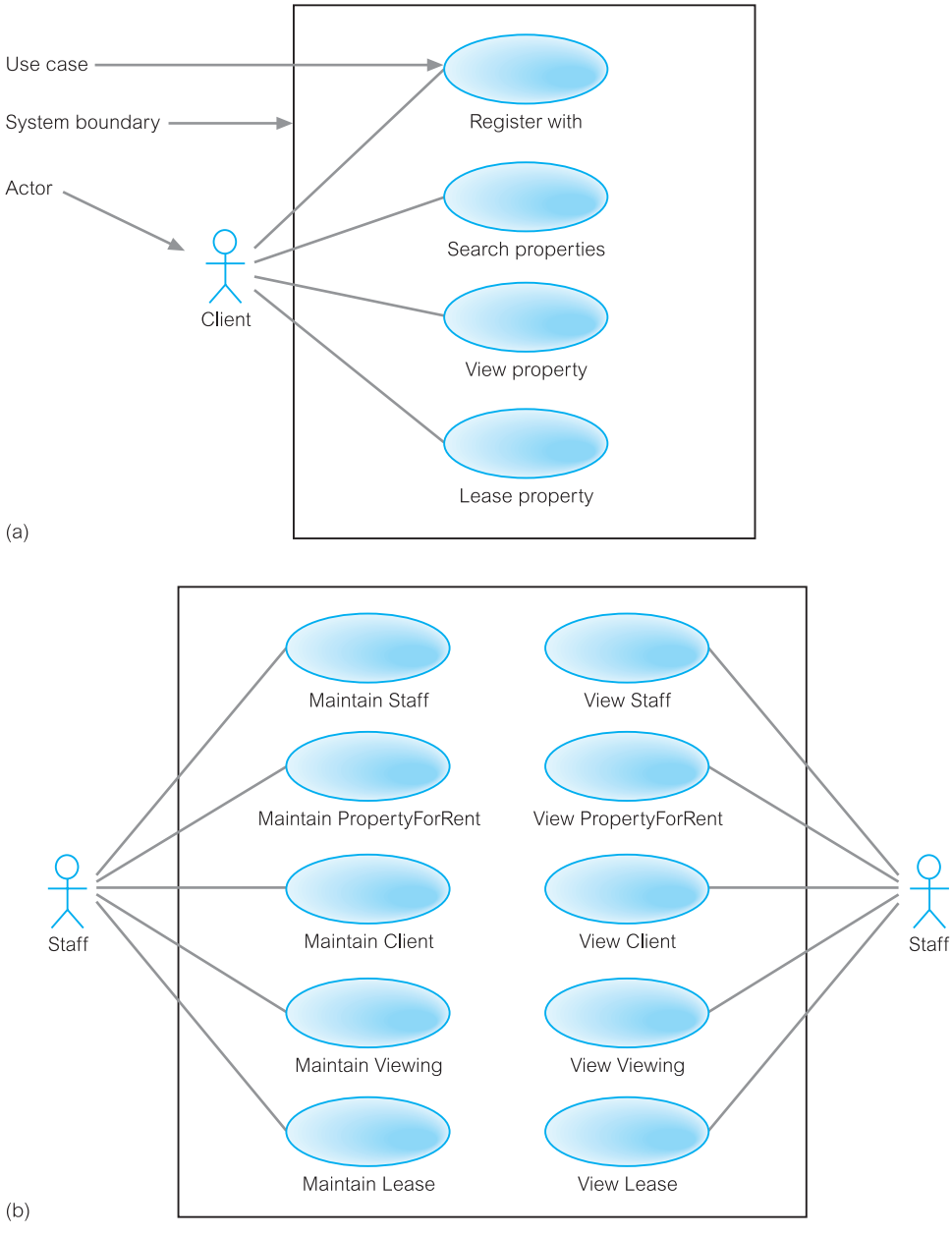


Figure 25.17
(a) Use case diagram with an actor (Client) and four use cases; (b) use case diagram for Staff.

which models the interaction in a column and row type format, the collaboration diagram uses the free-form arrangement of objects, which makes it easier to see all interactions involving a particular object. Messages are labeled with a chronological number to maintain ordering information. Figure 25.19 provides an example of a collaboration diagram for the Search properties use case.

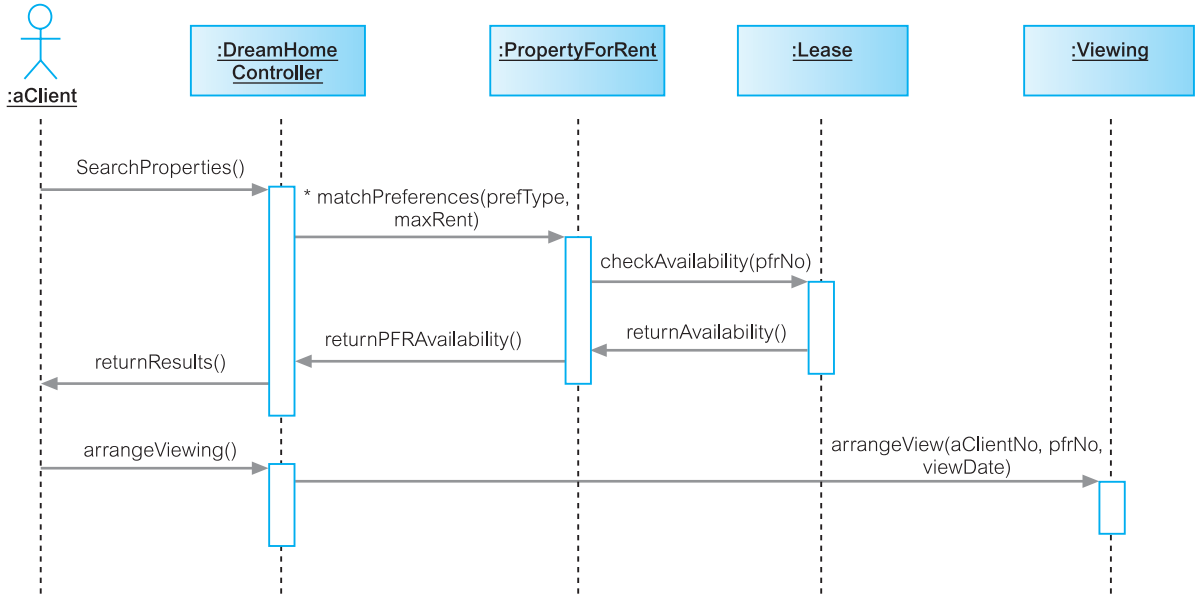


Figure 25.18 Sequence diagram for Search properties use case.

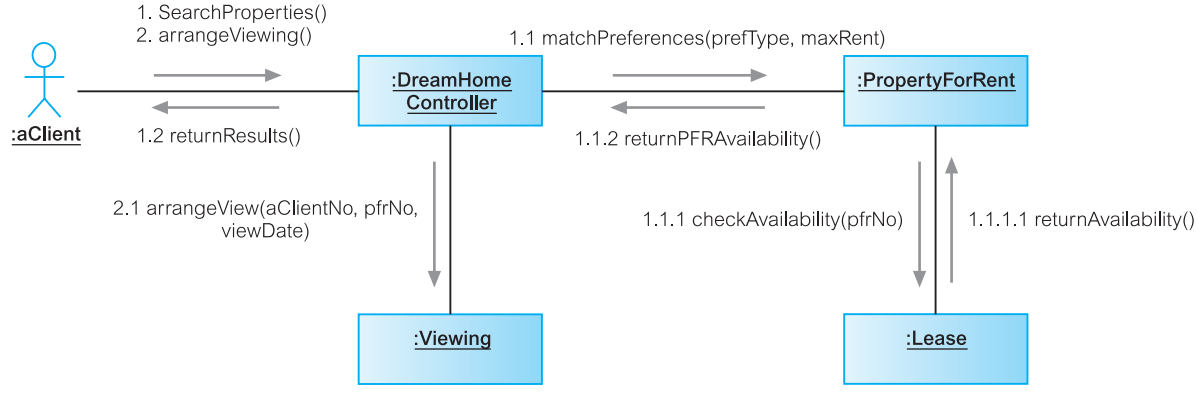


Figure 25.19 Collaboration diagram for Search properties use case.

Statechart diagrams

Statechart diagrams, sometimes referred to as state diagrams, show how objects can change in response to external events. While other behavioral diagrams typically model the interaction between multiple objects, statechart diagrams usually model the transitions of a specific object. Figure 25.20 provides an example of a statechart diagram for PropertyForRent. Again, the notation is simple consisting of a few symbols:

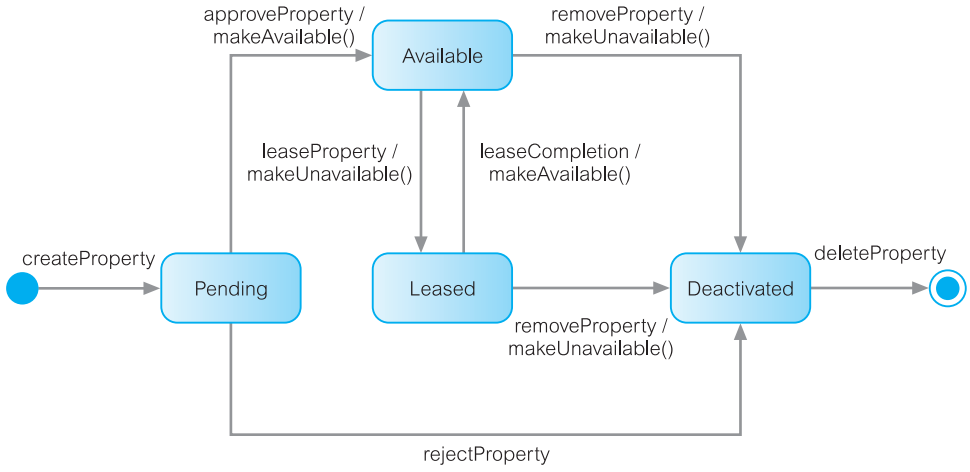


Figure 25.20
Statechart diagram for PropertyForRent.

- *States* are represented by boxes with rounded corners.
- *Transitions* are represented by solid arrows between states labeled with the 'event-name/action' (the *event* triggers the transition and *action* is the result of the transition). For example, in Figure 25.20, the transition from state Pending to Available is triggered by an approveProperty event and gives rise to the action called makeAvailable().
- *Initial state* (the state of the object before any transitions) is represented by a solid circle with an arrow to the initial state.
- *Final state* (the state that marks the destruction of the object) is represented by a solid circle with a surrounding circle and an arrow coming from a preceding state.

Activity diagrams

Activity diagrams model the flow of control from one activity to another. An activity diagram typically represents the invocation of an operation, a step in a business process, or an entire business process. It consists of activity states and transitions between them. The diagram shows flow of control and branches (small diamonds) can be used to specify alternative paths of transitions. Parallel flows of execution are represented by fork and join constructs (solid rectangles). *Swimlanes* can be used to separate independent areas. Figure 25.21 shows a first-cut activity diagram for *DreamHome*.

25.7.2 Usage of UML in the Methodology for Database Design

Many of the diagram types we have described above are useful during the database system development lifecycle, particularly during requirements collection and analysis, and database and application design. The following guidelines may prove helpful (McCready, 2003):

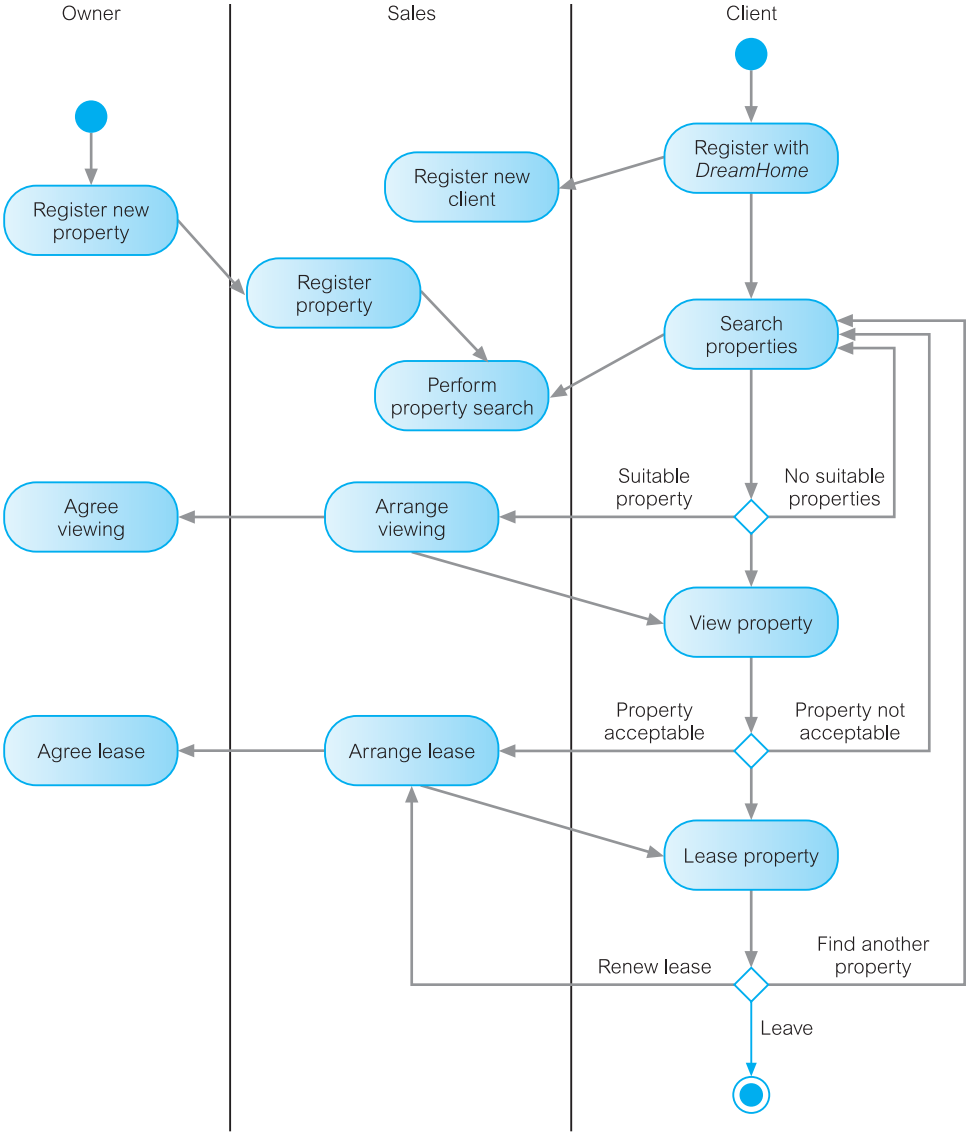


Figure 25.21 Sample activity diagram for *DreamHome*.

- Produce use case diagrams from the requirements specification or while producing the requirements specification to depict the main functions required of the system. The use cases can be augmented with use case descriptions, textual descriptions of each use case.
- Produce the first-cut class diagram (ER model).
- Produce a sequence diagram for each use case or group of related use cases. This will show the interaction between classes (entities) necessary to support the functionality defined in each use case. Collaboration diagrams can easily be produced from the

sequence diagrams (for example, the CASE tool Rational Rose can automatically produce a collaboration diagram from the corresponding sequence diagram).

- It may be useful to add a *control class* to the class diagram to represent the interface between the actors and the system (control class operations are derived from the use cases).
- Update the class diagram to show the required methods in each class.
- Create a state diagram for each class to show how the class changes state in response to messages it receives. The appropriate messages are identified from the sequence diagrams.
- Revise earlier diagrams based on new knowledge gained during this process (for example, the creation of state diagrams may identify additional methods for the class diagram).

Chapter Summary

- Advanced database applications include computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided software engineering (CASE), network management systems, office information systems (OIS) and multimedia systems, digital publishing, geographic information systems (GIS), and interactive and dynamic Web sites, as well as applications with complex and interrelated objects and procedural data.
- The relational model, and relational systems in particular, have weaknesses such as poor representation of 'real world' entities, semantic overloading, poor support for integrity and enterprise constraints, limited operations, and impedance mismatch. The limited modeling capabilities of relational DBMSs have made them unsuitable for advanced database applications.
- The concept of **encapsulation** means that an object contains both a data structure and the set of operations that can be used to manipulate it. The concept of **information hiding** means that the external aspects of an object are separated from its internal details, which are hidden from the outside world.
- An **object** is a uniquely identifiable entity that contains both the attributes that describe the **state** of a 'real world' object and the actions (**behavior**) that are associated with it. Objects can contain other objects. A key part of the definition of an object is unique identity. In an object-oriented system, each object has a unique system-wide identifier (the **OID**) that is independent of the values of its attributes and, ideally, invisible to the user.
- **Methods** define the behavior of the object. They can be used to change the object's state by modifying its attribute values or to query the value of selected attributes. **Messages** are the means by which objects communicate. A message is simply a request from one object (the sender) to another object (the receiver) asking the second object to execute one of its methods. The sender and receiver may be the same object.
- Objects that have the same attributes and respond to the same messages can be grouped together to form a **class**. The attributes and associated methods can then be defined once for the class rather than separately for each object. A class is also an object and has its own attributes and methods, referred to as **class attributes** and **class methods**, respectively. Class attributes describe the general characteristics of the class, such as totals or averages.

- **Inheritance** allows one class to be defined as a special case of a more general class. These special cases are known as **subclasses** and the more general cases are known as **superclasses**. The process of forming a superclass is referred to as **generalization**; forming a subclass is **specialization**. A subclass inherits all the properties of its superclass and additionally defines its own unique properties (attributes and methods). All instances of the subclass are also instances of the superclass. The *principle of substitutability* states that an instance of the subclass can be used whenever a method or a construct expects an instance of the superclass.
- **Overloading** allows the name of a method to be reused within a class definition or across definitions. **Overriding**, a special case of overloading, allows the name of a property to be redefined in a subclass. **Dynamic binding** allows the determination of an object's type and methods to be deferred until runtime.
- In response to the increasing complexity of database applications, two 'new' data models have emerged: the **Object-Oriented Data Model** (OODM) and the **Object-Relational Data Model** (ORDM). However, unlike previous models, the actual composition of these models is not clear. This evolution represents the **third generation** of DBMSs.

Review Questions

25.1 Discuss the general characteristics of advanced database applications.

25.2 Discuss why the weaknesses of the relational data model and relational DBMSs may make them unsuitable for advanced database applications.

25.3 Define each of the following concepts in the context of an object-oriented data model:
(a) abstraction, encapsulation, and information hiding;
(b) objects and attributes;
(c) object identity;
(d) methods and messages;
(e) classes, subclasses, superclasses, and inheritance;
(f) overriding and overloading;
(g) polymorphism and dynamic binding.
Give examples using the *DreamHome* sample data shown in Figure 3.3.

25.4 Discuss the difficulties involved in mapping objects created in an object-oriented programming language to a relational database.

25.5 Describe the three generations of DBMSs.

25.6 Describe how relationships can be modeled in an OODBMS.

25.7 Describe the different modeling notations in the UML.

Exercises

25.8 Investigate one of the advanced database applications discussed in Section 25.1, or a similar one that handles complex, interrelated data. In particular, examine its functionality and the data types and operations it uses. Map the data types and operations to the object-oriented concepts discussed in Section 25.3.

25.9 Analyze one of the RDBMSs that you currently use. Discuss the object-oriented features provided by the system. What additional functionality do these features provide?

25.10 For the *DreamHome* case study documented in Appendix A, suggest attributes and methods that would be appropriate for Branch, Staff, and PropertyForRent classes.

25.11 Produce use case diagrams and a set of associated sequence diagrams for the *DreamHome* case study documented in Appendix A.

25.12 Produce use case diagrams and a set of associated sequence diagrams for the *University Accommodation Office* case study documented in Appendix B.1.

25.13 Produce use case diagrams and a set of associated sequence diagrams for the *Easy Drive School of Motoring* case study documented in Appendix B.2.

25.14 Produce use case diagrams and a set of associated sequence diagrams for the *Wellmeadows Hospital* case study documented in Appendix B.3.

Chapter 26 Object-Oriented DBMSs – Concepts

Chapter Objectives

In this chapter you will learn:

- The framework for an object-oriented data model.
- The basics of the functional data model.
- The basics of persistent programming languages.
- The main points of the OODBMS Manifesto.
- The main strategies for developing an OODBMS.
- The difference between the two-level storage model used by conventional DBMSs and the single-level model used by OODBMSs.
- How pointer swizzling techniques work.
- The difference between how a conventional DBMS accesses a record and how an OODBMS accesses an object on secondary storage.
- The different schemes for providing persistence in programming languages.
- The advantages and disadvantages of orthogonal persistence.
- About various issues underlying OODBMSs, including extended transaction models, version management, schema evolution, OODBMS architectures, and benchmarking.
- The advantages and disadvantages of OODBMSs.

In the previous chapter we reviewed the weaknesses of the relational data model against the requirements for the types of advanced database applications that are emerging. We also introduced the concepts of object-orientation, which solve some of the classic problems of software development. Some of the advantages often cited in favor of object-orientation are:

- The definition of a system in terms of objects facilitates the construction of software components that closely resemble the application domain, thus assisting in the design and understandability of systems.
- Owing to encapsulation and information hiding, the use of objects and messages encourages modular design – the implementation of one object does not depend on the

internals of another, only on how it responds to messages. Further, modularity is reinforced and software can be made more reliable.

- The use of classes and inheritance promotes the development of reusable and extensible components in the construction of new or upgraded systems.

In this chapter we consider the issues associated with one approach to integrating object-oriented concepts with database systems, namely the **Object-Oriented Database Management System** (OODBMS). The OODBMS started in the engineering and design domains and has recently also become the favored system for financial and telecommunications applications. The OODBMS market is small in comparison to the relational DBMS market and while it had an estimated growth rate of 50% at the end of the 1990s, the market has not maintained this growth.

In the next chapter we examine the object model proposed by the Object Data Management Group, which has become a *de facto* standard for OODBMSs. We also look at ObjectStore, a commercial OODBMS.

Moving away from the traditional relational data model is sometimes referred to as a *revolutionary approach* to integrating object-oriented concepts with database systems. In contrast, in Chapter 28 we examine a more *evolutionary approach* to integrating object-oriented concepts with database systems that extends the relational model. These evolutionary systems are referred to now as **Object-Relational DBMSs** (ORDBMSs), although an earlier term used was *Extended-Relational DBMSs*.

Structure of this Chapter

In Section 26.1 we provide an introduction to object-oriented data models and persistent languages, and discuss how, unlike the relational data model, there is no universally agreed object-oriented data model. We also briefly review the *Object-Oriented Database System Manifesto*, which proposed thirteen mandatory features for an OODBMS, and examine the different approaches that can be taken to develop an OODBMS. In Section 26.2 we examine the difference between the two-level storage model used by conventional DBMSs and the single-level model used by OODBMSs, and how this affects data access. In Section 26.3 we discuss the various approaches to providing persistence in programming languages and the different techniques for pointer swizzling. In Section 26.4 we examine some other issues associated with OODBMSs, namely extended transaction models, version management, schema evolution, OODBMS architectures, and benchmarking. In Section 26.6 we review the advantages and disadvantages of OODBMSs.

To gain full benefit from this chapter, the reader needs to be familiar with the contents of Chapter 25. The examples in this chapter are once again drawn from the *DreamHome* case study documented in Section 10.4 and Appendix A.

26.1 Introduction to Object-Oriented Data Models and OODBMSs

In this section we discuss some background concepts to the OODBMS including the functional data model and persistent programming languages. We start by looking at the definition of an OODBMS.

26.1.1 Definition of Object-Oriented DBMSs

In this section we examine some of the different definitions that have been proposed for an object-oriented DBMS. Kim (1991) defines an Object-Oriented Data Model (OODM), Object-Oriented Database (OODB), and an Object-Oriented DBMS (OODBMS) as:

OODM	A (logical) data model that captures the semantics of objects supported in object-oriented programming.
OODB	A persistent and sharable collection of objects defined by an OODM.
OODBMS	The manager of an OODB.

These definitions are very non-descriptive and tend to reflect the fact that there is no one object-oriented data model equivalent to the underlying data model of relational systems. Each system provides its own interpretation of base functionality. For example, Zdonik and Maier (1990) present a threshold model that an OODBMS must, at a minimum, satisfy:

(1) it must provide database functionality;

(2) it must support object identity;

(3) it must provide encapsulation;

(4) it must support objects with complex state.

The authors argue that although inheritance may be useful, it is not essential to the definition, and an OODBMS could exist without it. On the other hand, Khoshafian and Abnous (1990) define an OODBMS as:

(1) object-orientation = abstract data types + inheritance + object identity;

(2) OODBMS = object-orientation + database capabilities.

Yet another definition of an OODBMS is given by Parsaye *et al.* (1989):

(1) high-level query language with query optimization capabilities in the underlying system;

(2) support for persistence, atomic transactions, and concurrency and recovery control;

(3) support for complex object storage, indexes, and access methods for fast and efficient retrieval;

(4) OODBMS = object-oriented system + (1) + (2) + (3).

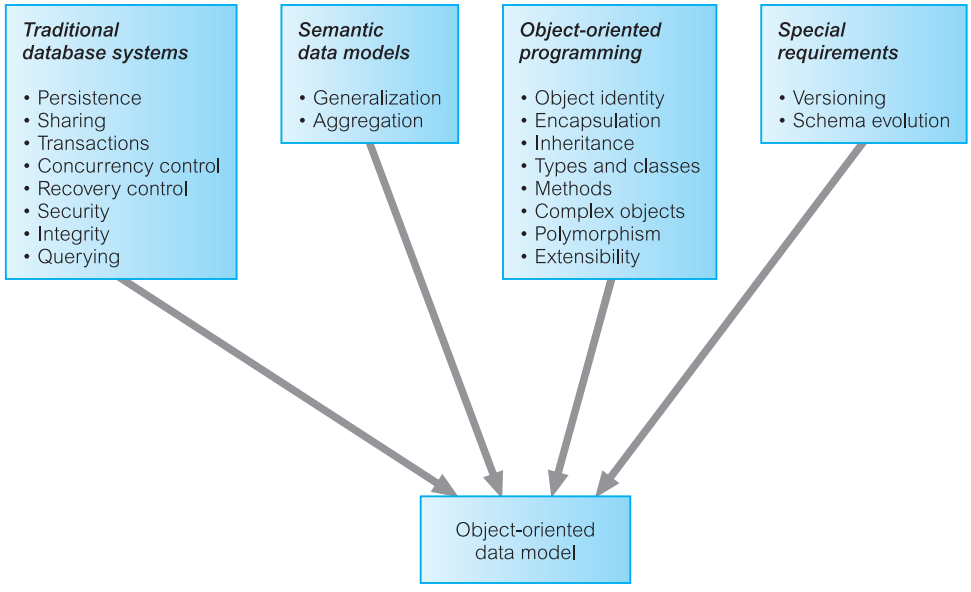


Figure 26.1
Origins of object-oriented data model.

Studying some of the current commercial OODBMSs, such as GemStone from Gemstone Systems Inc. (previously Servio Logic Corporation), Objectivity/DB from Objectivity Inc., ObjectStore from Progress Software Corporation (previously Object Design Inc.), 'FastObjects by Poet' from Poet Software Corporation, Jasmine Object Database from Computer Associates/Fujitsu Limited, and Versant (VDS) from Versant Corporation, we can see that the concepts of object-oriented data models are drawn from different areas, as shown in Figure 26.1.

In Section 27.2 we examine the object model proposed by the Object Data Management Group (ODMG), which many of these vendors intend to support. The ODMG object model is important because it specifies a standard model for the semantics of database objects and supports interoperability between compliant OODBMSs. For surveys of the basic concepts of Object-Oriented Data Models the interested reader is referred to Dittrich (1986) and Zaniola *et al.* (1986).

26.1.2 Functional Data Models

In this section we introduce the functional data model (FDM), which is one of the simplest in the family of semantic data models (Kerschberg and Pacheco, 1976; Sibley and Kerschberg, 1977). This model is interesting because it shares certain ideas with the object approach including object identity, inheritance, overloading, and navigational access. In the FDM, any data retrieval task can be viewed as the process of evaluating and returning the result of a function with zero, one, or more arguments. The resulting data model is conceptually simple while at the same time is very expressive. In the FDM, the main modeling primitives are **entities** and **functional relationships**.

Entities

Entities are decomposed into (abstract) entity types and printable entity types. **Entity types** correspond to classes of 'real world' objects and are declared as functions with zero arguments that return the type ENTITY. For example, we could declare the Staff and PropertyForRent entity types as follows:

Staff() → ENTITY
PropertyForRent() → ENTITY

Printable entity types are analogous to the base types in a programming language and include: INTEGER, CHARACTER, STRING, REAL, and DATE. An attribute is defined as a *functional relationship*, taking the entity type as an argument and returning a printable entity type. Some of the attributes of the Staff entity type could be declared as follows:

staffNo(Staff) → STRING
sex(Staff) → CHAR
salary(Staff) → REAL

Thus, applying the function staffNo to an entity of type Staff returns that entity's staff number, which is a printable value of type STRING. We can declare a composite attribute by first declaring the attribute to be an entity type and then declaring its components as functional relationships of the entity type. For example, we can declare the composite attribute Name of Staff as follows:

Name() → ENTITY
Name(Staff) → NAME
fName(Name) → STRING
lName(Name) → STRING

Relationships

Functions with arguments model not only the properties (attributes) of entity types but also relationships between entity types. Thus, the FDM makes no distinction between attributes and relationships. Each relationship may have an inverse relationship defined. For example, we may model the one-to-many relationship Staff *Manages* PropertyForRent as follows:

Manages(Staff) →» PropertyForRent
ManagedBy(PropertyForRent) → Staff INVERSE OF Manages

In this example, the double-headed arrow is used to represent a one-to-many relationship. This notation can also be used to represent multi-valued attributes. Many-to-many relationships can be modeled by using the double-headed arrow in both directions. For example, we may model the *.* relationship Client *Views* PropertyForRent as follows:

Views(Client) →» PropertyForRent
ViewedBy(PropertyForRent) →» Client INVERSE OF Views

Note, an entity (instance) is some form of token identifying a unique object in the database and typically representing a unique object in the 'real world'. In addition, a function maps a given entity to one or more target entities (for example, the function Manages maps a

particular Staff entity to a set of PropertyForRent entities). Thus, all inter-object relationships are modeled by associating the corresponding entity instances and not their names or keys. Thus, referential integrity is an implicit part of the functional data model and requires no explicit enforcement, unlike the relational data model.

The FDM also supports multi-valued functions. For example, we can model the attribute viewDate of the previous relationship Views as follows:

viewDate(Client, PropertyForRent) → DATE

Inheritance and path expressions

The FDM supports inheritance through entity types. For example, the function Staff() returns a set of staff entities formed as a subset of the ENTITY type. Thus, the entity type Staff is a subtype of the entity type ENTITY. This subtype/supertype relationship can be extended to any level. As would be expected, subtypes inherit all the functions defined over all of its supertypes. The FDM also supports the principle of substitutability (see Section 25.3.6), so that an instance of a subtype is also an instance of its supertypes. For example, we could declare the entity type Supervisor to be a subtype of the entity type Staff as follows:

Staff()→ ENTITY
Supervisor()→ ENTITY
IS-A-STAFF(Supervisor) → Staff

The FDM allows **derived functions** to be defined from the composition of multiple functions. Thus, we can define the following derived functions (note the overloading of function names):

fName(Staff) → fName(Name(Staff))
fName(Supervisor) → fName(IS-A-STAFF(Supervisor))

The first derived function returns the set of first names of staff by evaluating the composite function on the right-hand side of the definition. Following on from this, in the second case the right-hand side of the definition is evaluated as the composite function fName(Name(IS-A-STAFF(Supervisor))). This composition is called a **path expression** and may be more recognizable written in *dot notation*:

Supervisor.IS-A-STAFF.Name.fname

Figure 26.2(a) provides a declaration of part of the *DreamHome* case study as an FDM schema and Figure 26.2(b) provides a corresponding graphical representation.

Functional query languages

Path expressions are also used within a functional query language. We will not discuss query languages in any depth but refer the interested reader to the papers cited at the end of this section. Instead, we provide a simple example to illustrate the language. For example, to retrieve the surnames of clients who have viewed a property managed by staff member SG14, we could write:

RETRIEVE lName(Name(ViewedBy(Manages(Staff))))
WHERE staffNo(Staff) = 'SG14'

Working from the inside of the path expression outwards, the function Manages(Staff) returns a set of PropertyForRent entities. Applying the function ViewedBy to this result returns a set of Client entities. Finally, applying the functions Name and lName returns the surnames of these clients. Once again, the equivalent dot notation may be more recognizable:

RETRIEVE Staff.Manages.ViewedBy.Name.lName
WHERE Staff.staffNo = 'SG14'

Note, the corresponding SQL statement would require three joins and is less intuitive than the FDM statement:

SELECT c.lName
FROM Staff s, PropertyForRent p, Viewing v, Client c
WHERE s.staffNo = p.staffNo **AND** p.propertyNo = v.propertyNo **AND**
v.clientNo = c.clientNo **AND** s.staffNo = 'SG14'

Advantages

Some of the advantages of the FDM include:

- *Support for some object-oriented concepts* The FDM is capable of supporting object identity, inheritance through entity class hierarchies, function name overloading, and navigational access.
- *Support for referential integrity* The FDM is an entity-based data model and implicitly supports referential integrity.
- *Irreducibility* The FDM is composed of a small number of simple concepts that represent semantically irreducible units of information. This allows a database schema to be depicted graphically with relative ease thereby simplifying conceptual design.
- *Easy extensibility* Entity classes and functions can be added/deleted without requiring modification to existing schema objects.
- *Suitability for schema integration* The conceptual simplicity of the FDM means that it can be used to represent a number of different data models including relational, network, hierarchical, and object-oriented. This makes the FDM a suitable model for the integration of heterogeneous schemas within multidatabase systems (MDBSs) discussed in Section 22.1.3.
- *Declarative query language* The query language is declarative with well-understood semantics (based on lambda calculus). This makes the language easy to transform and optimize.

There have been many proposals for functional data models and languages. The two earliest were FQL (Buneman and Frankel, 1979) and, perhaps the best known, DAPLEX (Shipman, 1981). The attraction of the functional style of these languages has produced many systems such as GDM (Batory *et al.*, 1988), the Extended FDM (Kulkarni and Atkinson, 1986, 1987), FDL (Poulovassilis and King, 1990), PFL (Poulovassilis and

Entity type declarations

Staff() → ENTITY
Supervisor() → ENTITY
PropertyForRent() → ENTITY
Client() → ENTITY
Name → ENTITY

Attribute declarations

fName(Name) → STRING
lName(Name) → STRING
fName(Staff) → fName(Name(Staff))
lName(Staff) → lName(Name(Staff))
fName(Client) → fName(Name(Client))
lName(Client) → lName(Name(Client))
staffNo(Staff) → STRING
position(Staff) → STRING
sex(Staff) → CHAR
salary(Staff) → REAL
clientNo(Client) → STRING
telNo(Client) → STRING
prefType(Client) → STRING
maxRent(Client) → REAL
propertyNo(PropertyForRent) → STRING
street(PropertyForRent) → STRING
city(PropertyForRent) → STRING
type(PropertyForRent) → STRING
rooms(PropertyForRent) → INTEGER
rent(PropertyForRent) → REAL

Relationship type declarations

Manages(Staff) →↠ PropertyForRent
ManagedBy(PropertyForRent) → Staff INVERSE OF Manages
Views(Client) →↠ PropertyForRent
ViewedBy(PropertyForRent) →↠ Client INVERSE OF Views
viewDate(Client, PropertyForRent) → DATE
comments(Client, PropertyForRent) → STRING

Inheritance declarations

IS-A-STAFF(Supervisor) → Staff
staffNo(Supervisor) → staffNo(IS-A-STAFF(Supervisor))
fName(Supervisor) → fName(IS-A-STAFF(Supervisor))
lName(Supervisor) → lName(IS-A-STAFF(Supervisor))
position(Supervisor) → position(IS-A-STAFF(Supervisor))
sex(Supervisor) → sex(IS-A-STAFF(Supervisor))
salary(Supervisor) → salary(IS-A-STAFF(Supervisor))

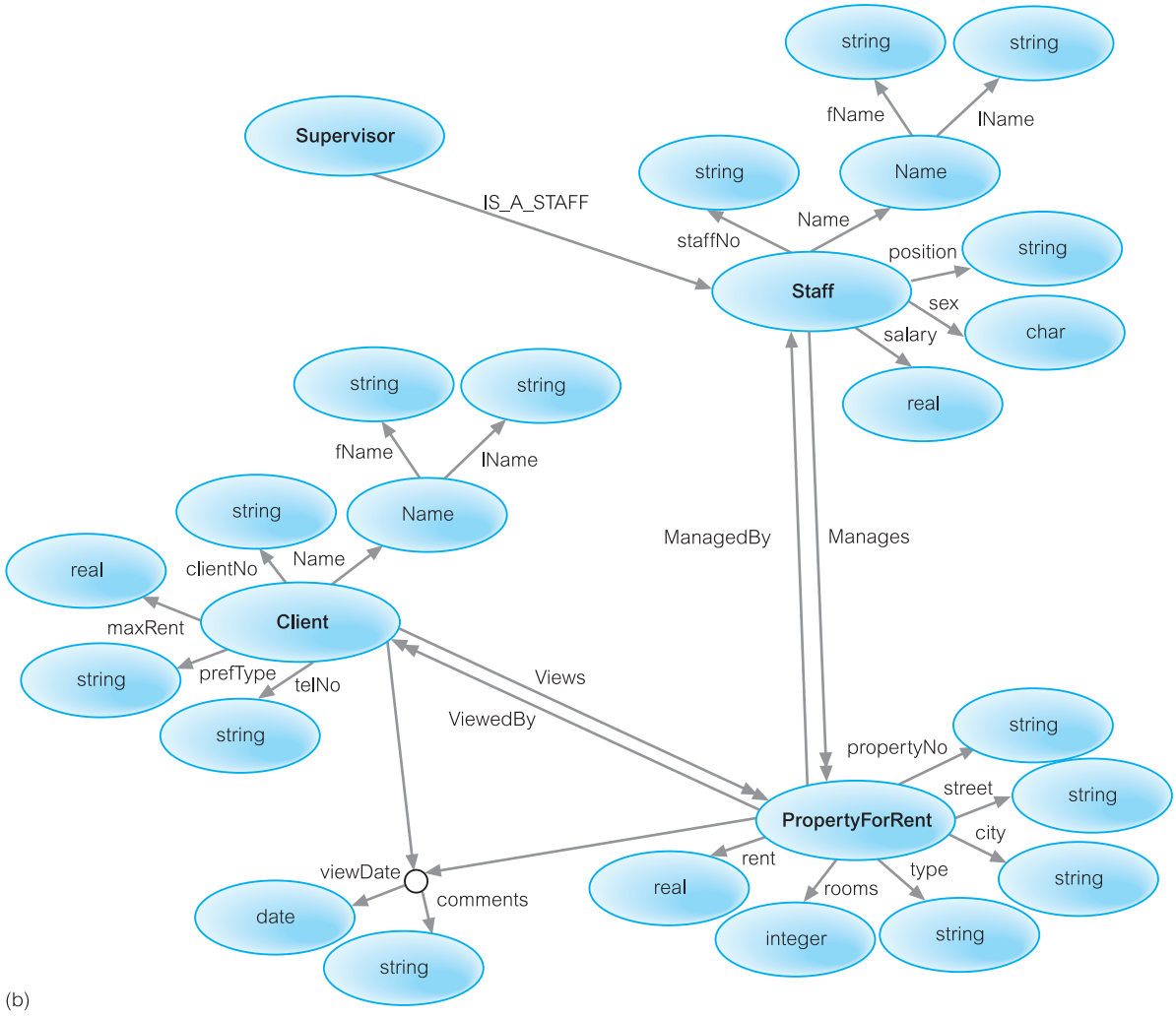
(a)

Figure 26.2 (a) Declaration of part of *DreamHome* as an FDM schema; (b) corresponding diagrammatic representation.

Small, 1991), and P/FDM (Gray *et al.*, 1992). The functional data languages have also been used with non-functional data models, such as PDM (Manola and Dayal, 1986), IPL (Annevelink, 1991), and LIFOO (Boucelma and Le Maitre, 1991). In the next section we examine another area of research that played a role in the development of the OODBMS.

26.1.3 Persistent Programming Languages

Before we start to examine OODBMSs in detail, we introduce another interesting but separate area of development known as *persistent programming languages*.



(b)

Figure 26.2 (*cont'd*)

Persistent programming language	A language that provides its users with the ability to (transparently) preserve data across successive executions of a program, and even allows such data to be used by many different programs.

Data in a persistent programming language is independent of any program, able to exist beyond the execution and lifetime of the code that created it. Such languages were originally intended to provide neither full database functionality nor access to data from multiple languages (Cattell, 1994).

Database programming language	A language that integrates some ideas from the database programming model with traditional programming language features.

In contrast, a database programming language is distinguished from a persistent programming language by its incorporation of features beyond persistence, such as transaction management, concurrency control, and recovery (Bancilhon and Buneman, 1990). The ISO SQL standard specifies that SQL can be embedded in the programming languages 'C', Fortran, Pascal, COBOL, Ada, MUMPS, and PL/1 (see Appendix E). Communication is through a set of variables in the host language, and a special preprocessor modifies the source code to replace the SQL statements with calls to DBMS routines. The source code can then be compiled and linked in the normal way. Alternatively, an API can be provided, removing the need for any precompilation. Although the embedded approach is rather clumsy, it was useful and necessary, as the SQL2 standard was not computationally complete.[†] The problems with using two different language paradigms have been collectively called the **impedance mismatch** between the application programming language and the database query language (see Section 25.2). It has been claimed that as much as 30% of programming effort and code space is devoted to converting data from database or file formats into and out of program-internal formats (Atkinson *et al.*, 1983). The integration of persistence into the programming language frees the programmer from this responsibility.

Researchers working on the development of persistent programming languages have been motivated primarily by the following aims (Morrison *et al.*, 1994):

- improving programming productivity by using simpler semantics;
- removing *ad hoc* arrangements for data translation and long-term data storage;
- providing protection mechanisms over the whole environment.

Persistent programming languages attempt to eliminate the impedance mismatch by extending the programming language with database capabilities. In a persistent programming language, the language's type system provides the data model, which usually contains rich structuring mechanisms. In some languages, for example PS-algol and Napier88, procedures are 'first class' objects and are treated like any other data objects in the language. For example, procedures are assignable, may be the result of expressions, other procedures or blocks, and may be elements of constructor types. Among other things, procedures can be used to implement abstract data types. The act of importing an abstract data type from the persistent store and dynamically binding it into a program is equivalent to module-linking in more traditional languages.

The second important aim of a persistent programming language is to maintain the same data representation in the application memory space as in the persistent store on secondary storage. This overcomes the difficulty and overhead of mapping between the two representations, as we see in Section 26.2.

The addition of (transparent) persistence into a programming language is an important enhancement to an interactive development environment, and the integration of the two paradigms provides increased functionality and semantics. The research into persistent programming languages has had a significant influence on the development of OODBMSs, and many of the issues that we discuss in Sections 26.2, 26.3, and 26.4 apply to both persistent programming languages and OODBMSs. The more encompassing term

[†] The 1999 release of the SQL standard, SQL:1999, added constructs to the language to make it computationally complete.

Persistent Application System (PAS) is sometimes used now instead of persistent programming language (Atkinson and Morrison, 1995).

The *Object-Oriented Database System Manifesto* 26.1.4

The 1989 *Object-Oriented Database System Manifesto* proposed thirteen mandatory features for an OODBMS, based on two criteria: it should be an object-oriented system and it should be a DBMS (Atkinson *et al*., 1989). The rules are summarized in Table 26.1. The first eight rules apply to the object-oriented characteristic.

(1) Complex objects must be supported

It must be possible to build complex objects by applying constructors to basic objects. The minimal set of constructors are SET, TUPLE, and LIST (or ARRAY). The first two are important because they have gained widespread acceptance as object constructors in the relational model. The final one is important because it allows order to be modeled. Furthermore, the manifesto requires that object constructors must be orthogonal: any constructor should apply to any object. For example, we should be able to use not only SET(TUPLE()) and LIST(TUPLE()) but also TUPLE(SET()) and TUPLE(LIST()).

(2) Object identity must be supported

All objects must have a unique identity that is independent of its attribute values.

(3) Encapsulation must be supported

In an OODBMS, proper encapsulation is achieved by ensuring that programmers have access only to the interface specification of methods, and the data and implementation of these methods are hidden in the objects. However, there may be cases where the enforcement

Table 26.1 Mandatory features in the *Object-Oriented Database System Manifesto*.

Object-oriented characteristics	DBMS characteristics
Complex objects must be supported	Data persistence must be provided
Object identity must be supported	The DBMS must be capable of handling very large databases
Encapsulation must be supported	The DBMS must support concurrent users
Types or classes must be supported	The DBMS must be capable of recovery from hardware and software failures
Types or classes must be able to inherit from their ancestors	The DBMS must provide a simple way of querying data
Dynamic binding must be supported	
The DML must be computationally complete	
The set of data types must be extensible	

of encapsulation is not required: for example, with *ad hoc* queries. (In Section 25.3.1 we noted that encapsulation is seen as one of the great strengths of the object-oriented approach. In which case, why should there be situations where encapsulation can be overriden? The typical argument given is that it is not an ordinary user who is examining the contents of objects but the DBMS. Second, the DBMS could invoke the 'get' method associated with every attribute of every class, but direct examination is more efficient. We leave these arguments for the reader to reflect on.)

(4) Types or classes must be supported

We mentioned the distinction between types and classes in Section 25.3.5. The manifesto requires support for only one of these concepts. The database schema in an object-oriented system comprises a set of classes or a set of types. However, it is not a requirement that the system automatically maintain the *extent* of a type, that is, the set of objects of a given type in the database, or if an extent is maintained, that the system should make it accessible to the user.

(5) Types or classes must be able to inherit from their ancestors

A subtype or subclass should inherit attributes and methods from its supertype or superclass, respectively.

(6) Dynamic binding must be supported

Methods should apply to objects of different types (overloading). The implementation of a method will depend on the type of the object it is applied to (overriding). To provide this functionality, the system cannot bind method names until runtime (dynamic binding).

(7) The DML must be computationally complete

In other words, the Data Manipulation Language (DML) of the OODBMS should be a general-purpose programming language. This was obviously not the case with the SQL2 standard (see Section 5.1), although with the release of the SQL:1999 standard the language is computationally complete (see Section 28.4).

(8) The set of data types must be extensible

The user must be able to build new types from the set of predefined system types. Furthermore, there must be no distinction in usage between system-defined and user-defined types.

The final five mandatory rules of the manifesto apply to the DBMS characteristic of the system.

(9) Data persistence must be provided

As in a conventional DBMS, data must remain (persist) after the application that created it has terminated. The user should not have to explicitly move or copy data to make it persistent.

(10) The DBMS must be capable of managing very large databases

In a conventional DBMS, there are mechanisms to manage secondary storage efficiently, such as indexes and buffers. An OODBMS should have similar mechanisms that are invisible to the user, thus providing a clear independence between the logical and physical levels of the system.

(11) The DBMS must support concurrent users

An OODBMS should provide concurrency control mechanisms similar to those in conventional systems.

(12) The DBMS must be capable of recovery from hardware and software failures

An OODBMS should provide recovery mechanisms similar to those in conventional systems.

(13) The DBMS must provide a simple way of querying data

An OODBMS must provide an *ad hoc* query facility that is high-level (that is, reasonably declarative), efficient (suitable for query optimization), and application-independent. It is not necessary for the system to provide a query language; it could instead provide a graphical browser.

The manifesto proposes the following optional features: multiple inheritance, type checking and type inferencing, distribution across a network, design transactions, and versions. Interestingly, there is no direct mention of support for security, integrity, or views; even a fully declarative query language is not mandated.

Alternative Strategies for Developing an OODBMS 26.1.5

There are several approaches to developing an OODBMS, which can be summarized as follows (Khoshafian and Abnous, 1990):

- *Extend an existing object-oriented programming language with database capabilities* This approach adds traditional database capabilities to an existing object-oriented programming language such as Smalltalk, C++, or Java (see Figure 26.1). This is the approach taken by the product GemStone, which extends these three languages.
- *Provide extensible object-oriented DBMS libraries* This approach also adds traditional database capabilities to an existing object-oriented programming language. However, rather than extending the language, class libraries are provided that support persistence, aggregation, data types, transactions, concurrency, security, and so on. This is the approach taken by the products Ontos, Versant, and ObjectStore. We discuss ObjectStore in Section 27.3.

- *Embed object-oriented database language constructs in a conventional host language* In Appendix E we describe how SQL can be embedded in a conventional host programming language. This strategy uses the same idea of embedding an object-oriented database language in a host programming language. This is the approach taken by O_2, which provided embedded extensions for the programming language 'C'.
- *Extend an existing database language with object-oriented capabilities* Owing to the widespread acceptance of SQL, vendors are extending it to provide object-oriented constructs. This approach is being pursued by both RDBMS and OODBMS vendors. The 1999 release of the SQL standard, SQL:1999, supports object-oriented features. (We review these features in Section 28.4.) In addition, the Object Database Standard by the Object Data Management Group (ODMG) specifies a standard for Object SQL, which we discuss in Section 27.2.4. The products Ontos and Versant provide a version of Object SQL and many OODBMS vendors will comply with the ODMG standard.
- *Develop a novel database data model/data language* This is a radical approach that starts from the beginning and develops an entirely new database language and DBMS with object-oriented capabilities. This is the approach taken by SIM (Semantic Information Manager), which is based on the semantic data model and has a novel DML/DDL (Jagannathan *et al*., 1988).

26.2 OODBMS Perspectives

DBMSs are primarily concerned with the creation and maintenance of large, long-lived collections of data. As we have already seen from earlier chapters, modern DBMSs are characterized by their support of the following features:

- *A data model* A particular way of describing data, relationships between data, and constraints on the data.
- *Data persistence* The ability for data to outlive the execution of a program and possibly the lifetime of the program itself.
- *Data sharing* The ability for multiple applications (or instances of the same one) to access common data, possibly at the same time.
- *Reliability* The assurance that the data in the database is protected from hardware and software failures.
- *Scalability* The ability to operate on large amounts of data in simple ways.
- *Security and integrity* The protection of the data against unauthorized access, and the assurance that the data conforms to specified correctness and consistency rules.
- *Distribution* The ability to physically distribute a logically interrelated collection of shared data over a computer network, preferably making the distribution transparent to the user.

In contrast, traditional programming languages provide constructs for procedural control and for data and functional abstraction, but lack built-in support for many of the above database features. While each is useful in its respective domain, there exists an increasing number of applications that require functionality from both DBMSs and programming

languages. Such applications are characterized by their need to store and retrieve large amounts of shared, structured data, as discussed in Section 25.1. Since 1980 there has been considerable effort expended in developing systems that integrate the concepts from these two domains. However, the two domains have slightly different perspectives that have to be considered and the differences addressed.

Perhaps two of the most important concerns from the programmers' perspective are performance and ease of use, both achieved by having a more seamless integration between the programming language and the DBMS than that provided with traditional DBMSs. With a traditional DBMS, we find that:

- It is the programmer's responsibility to decide when to read and update objects (records).
- The programmer has to write code to translate between the application's object model and the data model of the DBMS (for example, relations), which might be quite different. With an object-oriented programming language, where an object may be composed of many subobjects represented by pointers, the translation may be particularly complex. As noted above, it has been claimed that as much as 30% of programming effort and code space is devoted to this type of mapping. If this mapping process can be eliminated or at least reduced, the programmer would be freed from this responsibility, the resulting code would be easier to understand and maintain, and performance may increase as a result.
- It is the programmer's responsibility to perform additional type-checking when an object is read back from the database. For example, the programmer may create an object in the strongly typed object-oriented language Java and store it in a traditional DBMS. However, another application written in a different language may modify the object, with no guarantee that the object will conform to its original type.

These difficulties stem from the fact that conventional DBMSs have a two-level storage model: the application storage model in main or virtual memory, and the database storage model on disk, as illustrated in Figure 26.3. In contrast, an OODBMS tries to give the illusion of a single-level storage model, with a similar representation in both memory and in the database stored on disk, as illustrated in Figure 26.4.

Although the single-level memory model looks intuitively simple, the OODBMS has to cleverly manage the representations of objects in memory and on disk to achieve this illusion. As we discussed in Section 25.3 objects, and relationships between objects, are identified by object identifiers (OIDs). There are two types of OID:

- logical OIDs that are independent of the physical location of the object on disk;
- physical OIDs that encode the location.

In the former case, a level of indirection is required to look up the physical address of the object on disk. In both cases, however, an OID is different in size from a standard in-memory pointer that need only be large enough to address all virtual memory. Thus, to achieve the required performance, an OODBMS must be able to convert OIDs to and from in-memory pointers. This conversion technique has become known as **pointer swizzling** or **object faulting**, and the approaches used to implement it have become varied, ranging from software-based residency checks to page faulting schemes used by the underlying hardware (Moss and Eliot, 1990), as we now discuss.

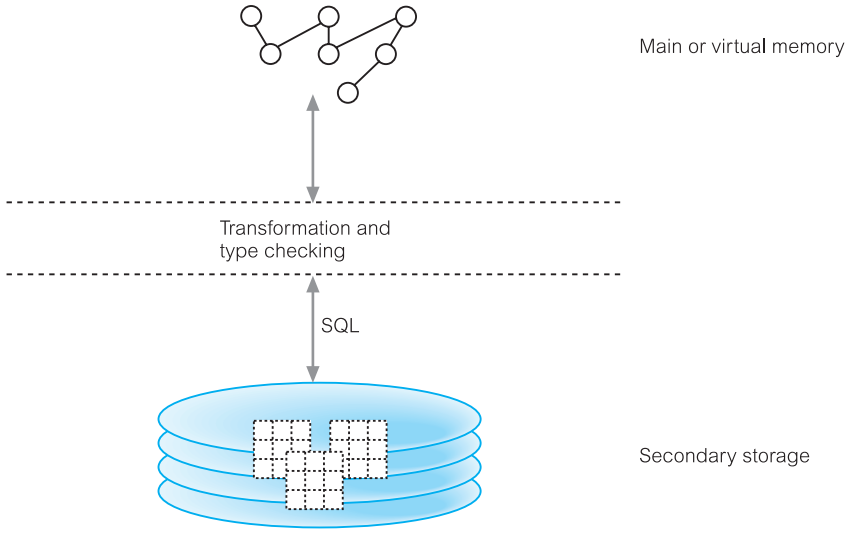


Figure 26.3
Two-level storage model for conventional (relational) DBMS.

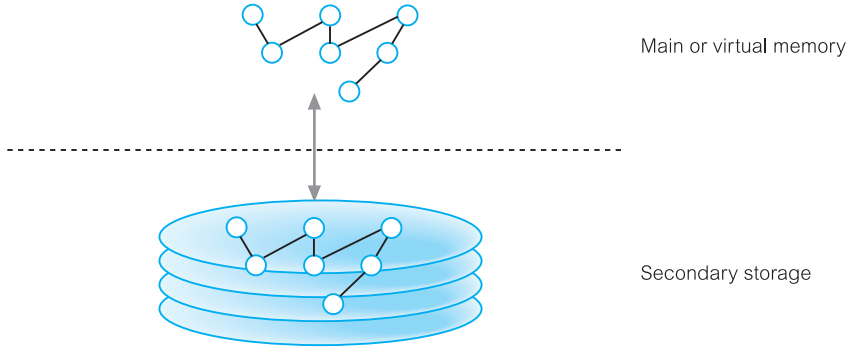


Figure 26.4
Single-level storage model for OODBMS.

26.2.1 Pointer Swizzling Techniques

Pointer swizzling	The action of converting object identifiers to main memory pointers, and back again.

The aim of pointer swizzling is to optimize access to objects. As we have just mentioned, references between objects are normally represented using OIDs. If we read an object from secondary storage into the page cache, we should be able to locate any referenced objects on secondary storage using their OIDs. However, once the referenced objects have been read into the cache, we want to record that these objects are now held in main memory to prevent them being retrieved from secondary storage again. One approach is to hold a lookup table that maps OIDs to main memory pointers. We can implement the table lookup reasonably efficiently using hashing, but this is still slow compared to a pointer

dereference, particularly if the object is already in memory. However, pointer swizzling attempts to provide a more efficient strategy by storing the main memory pointers in place of the referenced OIDs and vice versa when the object has to be written back to disk.

In this section we describe some of the issues surrounding pointer swizzling, including the various techniques that can be employed.

No swizzling

The easiest implementation of faulting objects into and out of memory is not to do any swizzling at all. In this case, objects are faulted into memory by the underlying object manager and a handle is passed back to the application containing the object's OID (White, 1994). The OID is used every time the object is accessed. This requires that the system maintain some type of lookup table so that the object's virtual memory pointer can be located and then used to access the object. As the lookup is required on each object access, this approach could be inefficient if the same object is accessed repeatedly. On the other hand, if an application tends only to access an object once, then this could be an acceptable approach.

Figure 26.5 shows the contents of the lookup table, sometimes called the **Resident Object Table** (**ROT**), after four objects have been read from secondary storage. If we now wish to access the Staff object with object identity OID5 from the Branch object OID1, a lookup of the ROT would indicate that the object was not in main memory and we would need to read the object from secondary storage and enter its memory address in the ROT table. On the other hand, if we try to access the Staff object with object identity OID4 from the Branch object, a lookup of the ROT would indicate that the object was already in main memory and provide its memory address.

Moss proposed an analytical model for evaluating the conditions under which swizzling is appropriate (1990). The results found suggest that if objects have a significant chance of being swapped out of main memory, or references are not followed at least several times

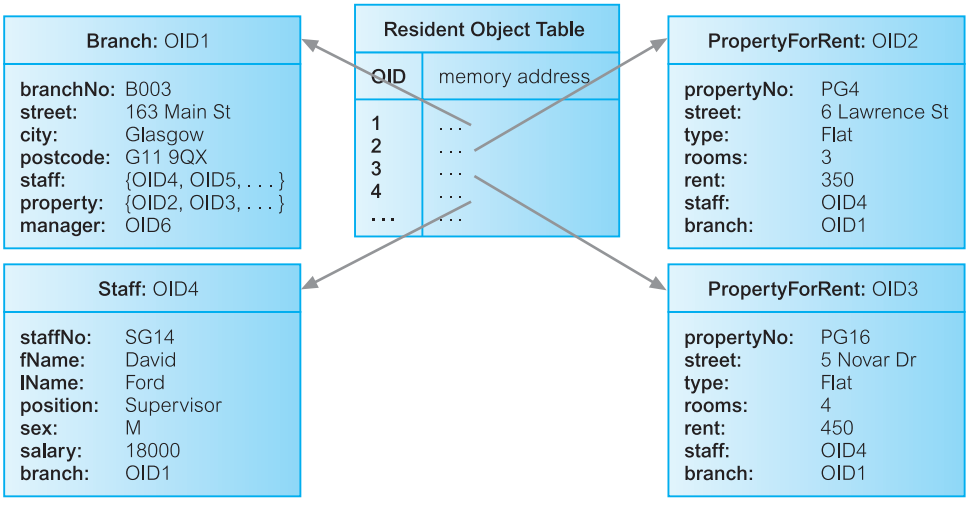


Figure 26.5
Resident Object Table referencing four objects in main memory.

on average, then an application would be better using efficient tables to map OIDs to object memory addresses (as in Objectivity/DB) rather than swizzling.

Object referencing

To be able to swizzle a persistent object's OID to a virtual memory pointer, a mechanism is required to distinguish between resident and non-resident objects. Most techniques are variations of either **edge marking** or **node marking** (Hoskings and Moss, 1993).

Considering virtual memory as a directed graph consisting of objects as nodes and references as directed edges, edge marking marks every object pointer with a tag bit. If the bit is set, then the reference is to a virtual memory pointer; otherwise, it is still pointing to an OID and needs to be swizzled when the object it refers to is faulted into the application's memory space. Node marking requires that all object references are immediately converted to virtual memory pointers when the object is faulted into memory. The first approach is a software-based technique but the second approach can be implemented using software- or hardware-based techniques.

In our previous example, the system replaces the value OID4 in the Branch object OID1 by its main memory address when Staff object OID4 is read into memory. This memory address provides a pointer that leads to the memory location of the Staff object identified by OID4. Thus, the traversal from Branch object OID1 to Staff object OID4 does not incur the cost of looking up an entry in the ROT, but consists now of a pointer dereference operation.

Hardware-based schemes

Hardware-based swizzling uses virtual memory access protection violations to detect accesses to non-resident objects (Lamb *et al.*, 1991). These schemes use the standard virtual memory hardware to trigger the transfer of persistent data from disk to main memory. Once a page has been faulted in, objects are accessed on that page via normal virtual memory pointers and no further object residency checking is required. The hardware approach has been used in several commercial and research systems including ObjectStore and Texas (Singhal *et al.*, 1992).

The main advantage of the hardware-based approach is that accessing memory-resident persistent objects is just as efficient as accessing transient objects because the hardware approach avoids the overhead of residency checks incurred by software approaches. A disadvantage of the hardware-based approach is that it makes the provision of many useful kinds of database functionality much more difficult, such as fine-grained locking, referential integrity, recovery, and flexible buffer management policies. In addition, the hardware approach limits the amount of data that can be accessed during a transaction to the size of virtual memory. This limitation could be overcome by using some form of garbage collection to reclaim memory space, although this would add overhead and complexity to the system.

Classification of pointer swizzling

Pointer swizzling techniques can be classified according to the following three dimensions:

(1) Copy versus in-place swizzling.
(2) Eager versus lazy swizzling.
(3) Direct versus indirect swizzling.

Copy versus in-place swizzling

When faulting objects in, the data can either be copied into the application's local object cache or it can be accessed in place within the object manager's page cache (White, 1994). As discussed in Section 20.3.4, the unit of transfer from secondary storage to the cache is the page, typically consisting of many objects. Copy swizzling may be more efficient as, in the worst case, only modified objects have to be swizzled back to their OIDs, whereas an in-place technique may have to unswizzle an entire page of objects if one object on the page is modified. On the other hand, with the copy approach, every object must be explicitly copied into the local object cache, although this does allow the page of the cache to be reused.

Eager versus lazy swizzling

Moss and Eliot (1990) define eager swizzling as the swizzling of all OIDs for persistent objects on all data pages used by the application before any object can be accessed. This is rather extreme, whereas Kemper and Kossman (1993) provide a more relaxed definition, restricting the swizzling to all persistent OIDs within the object the application wishes to access. Lazy swizzling swizzles pointers only as they are accessed or discovered. Lazy swizzling involves less overhead when an object is faulted into memory, but it does mean that two different types of pointer must be handled for every object access: a swizzled pointer and an unswizzled pointer.

Direct versus indirect swizzling

This is an issue only when it is possible for a swizzled pointer to refer to an object that is no longer in virtual memory. With direct swizzling, the virtual memory pointer of the referenced object is placed directly in the swizzled pointer; with indirect swizzling, the virtual memory pointer is placed in an intermediate object, which acts as a placeholder for the actual object. Thus, with the indirect scheme objects can be uncached without requiring the swizzled pointers that reference the object to be unswizzled also.

These techniques can be combined to give eight possibilities (for example, in-place/eager/direct, in-place/lazy/direct, or copy/lazy/indirect).

Accessing an Object 26.2.2

How an object is accessed on secondary storage is another important aspect that can have a significant impact on OODBMS performance. Again, if we look at the approach taken

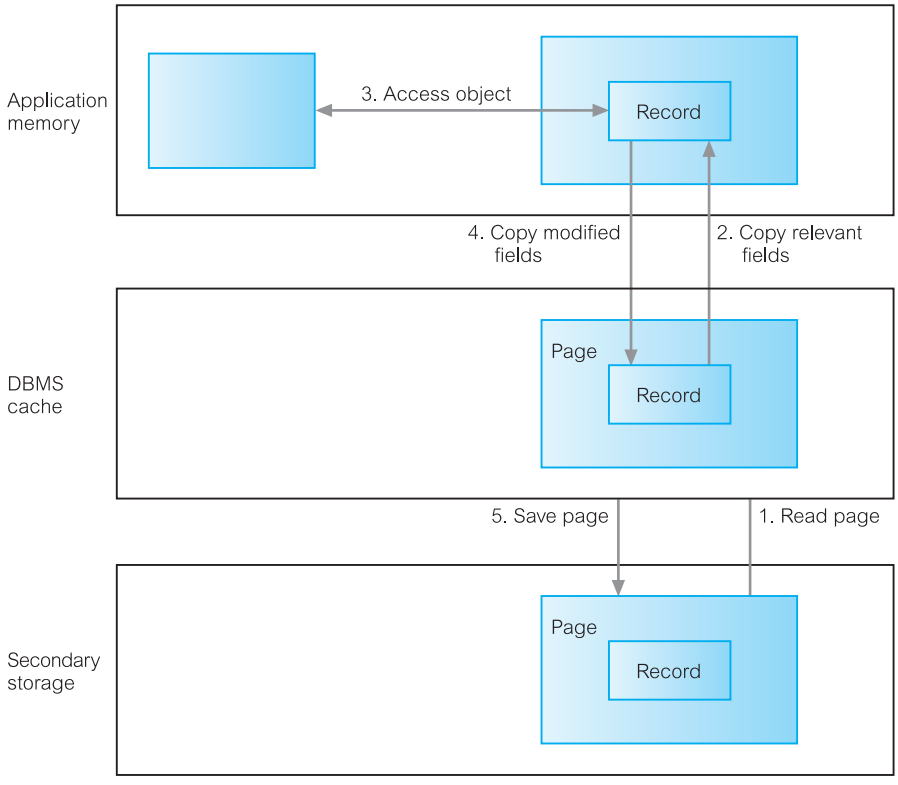


Figure 26.6
Steps in accessing a record using a conventional DBMS.

in a conventional relational DBMS with a two-level storage model, we find that the steps illustrated in Figure 26.6 are typical:

- The DBMS determines the page on secondary storage that contains the required record using indexes or table scans, as appropriate (see Section 21.4). The DBMS then reads that page from secondary storage and copies it into its cache.
- The DBMS subsequently transfers the required parts of the record from the cache into the application's memory space. Conversions may be necessary to convert the SQL data types into the application's data types.
- The application can then update the record's fields in its own memory space.
- The application transfers the modified fields back to the DBMS cache using SQL, again requiring conversions between data types.
- Finally, at an appropriate point the DBMS writes the updated page of the cache back to secondary storage.

In contrast, with a single-level storage model, an OODBMS uses the following steps to retrieve an object from secondary storage, as illustrated in Figure 26.7:

- The OODBMS determines the page on secondary storage that contains the required object using its OID or an index, as appropriate. The OODBMS then reads that

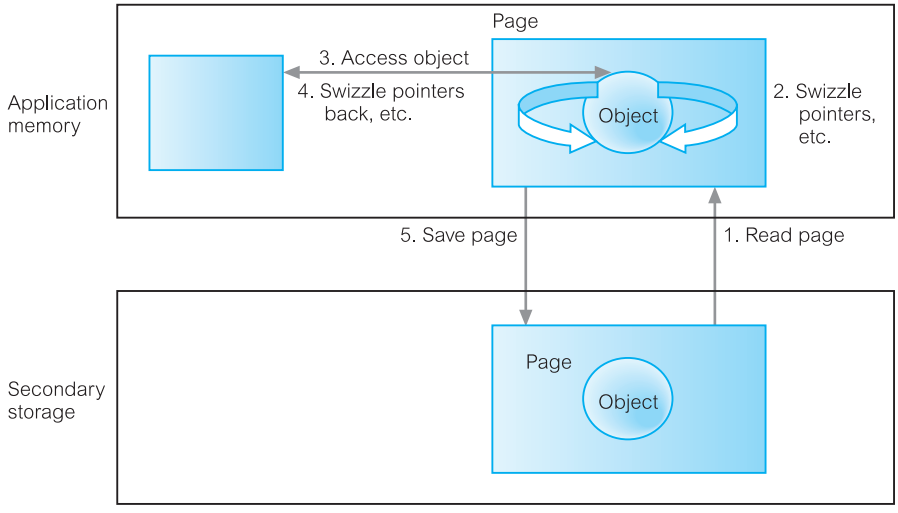


Figure 26.7
Steps in accessing an object using an OODBMS.

page from secondary storage and copies it into the application's page cache within its memory space.

- The OODBMS may then carry out a number of conversions, such as:
  - swizzling references (pointers) between objects;
  - adding some information to the object's data structure to make it conform to that required by the programming language;
  - modifying the data representations for data that has come from a different hardware platform or programming language.
- The application can then directly access the object and update it, as required.
- When the application wishes to make the changes persistent, or when the OODBMS needs to swap the page out of the page cache, the OODBMS may need to carry out similar conversions as listed above, before copying the page back to secondary storage.

26.3 Persistence

A DBMS must provide support for the storage of **persistent** objects, that is, objects that survive after the user session or application program that created them has terminated. This is in contrast to **transient** objects that last only for the invocation of the program. Persistent objects are retained until they are no longer required, at which point they are deleted. Other than the embedded language approach discussed in Section 26.1.3, the schemes we present next may be used to provide persistence in programming languages. For a complete survey of persistence schemes, the interested reader is referred to Atkinson and Buneman (1989).

Although intuitively we might consider persistence to be limited to the state of objects, persistence can also be applied to (object) code and to the program execution state. Including code in the persistent store potentially provides a more complete and elegant

solution. However, without a fully integrated development environment, making code persist leads to duplication, as the code will exist in the file system. Having program state and thread state persist is also attractive but, unlike code for which there is a standard definition of its format, program execution state is not easily generalized. In this section we limit our discussion to object persistence.

26.3.1 Persistence Schemes

In this section we briefly examine three schemes for implementing persistence within an OODBMS, namely checkpointing, serialization, and explicit paging.

Checkpointing

Some systems implement persistence by copying all or part of a program's address space to secondary storage. In cases where the complete address space is saved, the program can restart from the checkpoint. In other cases, only the contents of the program's heap are saved.

Checkpointing has two main drawbacks: typically, a checkpoint can be used only by the program that created it; second, a checkpoint may contain a large amount of data that is of no use in subsequent executions.

Serialization

Some systems implement persistence by copying the closure of a data structure to disk. In this scheme, a write operation on a data value typically involves the traversal of the graph of objects reachable from the value, and the writing of a flattened version of the structure to disk. Reading back this flattened data structure produces a new copy of the original data structure. This process is sometimes called **serialization**, **pickling**, or in a distributed computing context, **marshaling**.

Serialization has two inherent problems. First, it does not preserve object identity, so that if two data structures that share a common substructure are separately serialized, then on retrieval the substructure will no longer be shared in the new copies. Further, serialization is not incremental, and so saving small changes to a large data structure is not efficient.

Explicit paging

Some persistence schemes involve the application programmer explicitly 'paging' objects between the application heap and the persistent store. As discussed above, this usually requires the conversion of object pointers from a disk-based scheme to a memory-based scheme. With the explicit paging mechanism, there are two common methods for creating/updating persistent objects: reachability-based and allocation-based.

Reachability-based persistence means that an object will persist if it is reachable from a persistent root object. This method has some advantages including the notion that the programmer does not need to decide at object creation time whether the object should be persistent. At any time after creation, an object can become persistent by adding it to the

reachability tree. Such a model maps well on to a language such as Smalltalk or Java that contains some form of garbage collection mechanism, which automatically deletes objects when they are no longer accessible from any other object.

Allocation-based persistence means that an object is made persistent only if it is explicitly declared as such within the application program. This can be achieved in several ways, for example:

- *By class* A class is statically declared to be persistent and all instances of the class are made persistent when they are created. Alternatively, a class may be a subclass of a system-supplied persistent class. This is the approach taken by the products Ontos and Objectivity/DB.
- *By explicit call* An object may be specified as persistent when it is created or, in some cases, dynamically at runtime. This is the approach taken by the product ObjectStore. Alternatively, the object may be dynamically added to a persistent collection.

In the absence of pervasive garbage collection, an object will exist in the persistent store until it is explicitly deleted by the application. This potentially leads to storage leaks and dangling pointer problems.

With either of these approaches to persistence, the programmer needs to handle two different types of object pointer, which reduces the reliability and maintainability of the software. These problems can be avoided if the persistence mechanism is fully integrated with the application programming language, and it is this approach that we discuss next.

Orthogonal Persistence 26.3.2

An alternative mechanism for providing persistence in a programming language is known as **orthogonal persistence** (Atkinson *et al*., 1983; Cockshott, 1983), which is based on the following three fundamental principles.

Persistence independence

The persistence of a data object is independent of how the program manipulates that data object and conversely a fragment of a program is expressed independently of the persistence of data it manipulates. For example, it should be possible to call a function with its parameters sometimes objects with long-term persistence and at other times transient. Thus, the programmer does not need to (indeed cannot) program to control the movement of data between long- and short-term storage.

Data type orthogonality

All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where an object is not allowed to be long-lived or is not allowed to be transient. In some persistent languages, persistence is a quality attributable to only a subset of the language data types. This approach is exemplified by Pascal/R, Amber, Avalon/C++, and E. The orthogonal approach has been adopted by a number of systems, including PS-algol, Napier88, Galileo, and GemStone (Connolly, 1997).

Transitive persistence

The choice of how to identify and provide persistent objects at the language level is independent of the choice of data types in the language. The technique that is now widely used for identification is reachability-based, as discussed in the previous section. This principle was originally referred to as persistence identification but the more suggestive ODMG term 'transitive persistence' is used here.

Advantages and disadvantages of orthogonal persistence

The uniform treatment of objects in a system based on the principle of orthogonal persistence is more convenient for both the programmer and the system:

- there is no need to define long-term data in a separate schema language;
- no special application code is required to access or update persistent data;
- there is no limit to the complexity of the data structures that can be made persistent.

Consequently, orthogonal persistence provides the following advantages:

- improved programmer productivity from simpler semantics;
- improved maintenance – persistence mechanisms are centralized, leaving programmers to concentrate on the provision of business functionality;
- consistent protection mechanisms over the whole environment;
- support for incremental evolution;
- automatic referential integrity.

However, there is some runtime expense in a system where every pointer reference might be addressing a persistent object, as the system is required to test whether the object must be loaded from secondary storage. Further, although orthogonal persistence promotes transparency, a system with support for sharing among concurrent processes cannot be fully transparent.

Although the principles of orthogonal persistence are desirable, many OODBMSs do not implement them completely. There are some areas that require careful consideration and we briefly discuss two here, namely queries and transactions.

What objects do queries apply to?

From a traditional DBMS perspective, declarative queries range over persistent objects, that is, objects that are stored in the database. However, with orthogonal persistence we should treat persistent and transient objects in the same way. Thus, queries should range over both persistent and transient objects. But what is the scope for transient objects? Should the scope be restricted to the transient objects in the current user's run unit or should it also include the run units of other concurrent users? In either case, for efficiency we may wish to maintain indexes on transient as well as persistent objects. This may require some form of query processing within the client process in addition to the traditional query processing within the server.

What objects are part of transaction semantics?

From a traditional DBMS perspective, the ACID (Atomicity, Consistency, Isolation, and Durability) properties of a transaction apply to persistent objects (see Section 20.1.1). For example, whenever a transaction aborts, any updates that have been applied to persistent objects have to be undone. However, with orthogonal persistence we should treat persistent and transient objects in the same way. Thus, should the semantics of transactions apply also to transient objects? In our example, when we undo the updates to persistent objects should we also undo the changes to transient objects that have been made within the scope of the transaction? If this were the case, the OODBMS would have to log both the changes that are made to persistent objects and the changes that are made to transient objects. If a transient object were destroyed within a transaction, how would the OODBMS recreate this object within the user's run unit? There are a considerable number of issues that need to be addressed if transaction semantics range over both types of object. Unsurprisingly, few OODBMSs guarantee transaction consistency of transient objects.

26.4 Issues in OODBMSs

In Section 25.2 we mentioned three areas that are problematic for relational DBMSs, namely:

- long-duration transactions;
- versions;
- schema evolution.

In this section we discuss how these issues are addressed in OODBMSs. We also examine possible architectures for OODBMSs and briefly consider benchmarking.

26.4.1 Transactions

As discussed in Section 20.1, a transaction is a *logical unit of work*, which should always transform the database from one consistent state to another. The types of transaction found in business applications are typically of short duration. In contrast, transactions involving complex objects, such as those found in engineering and design applications, can continue for several hours, or even several days. Clearly, to support **long-duration transactions** we need to use different protocols from those used for traditional database applications in which transactions are typically of a very short duration.

In an OODBMS, the unit of concurrency control and recovery is logically an object, although for performance reasons a more coarse granularity may be used. Locking-based protocols are the most common type of concurrency control mechanism used by OODBMSs to prevent conflict from occurring. However, it would be totally unacceptable for a user who initiated a long-duration transaction to find that the transaction has been aborted owing to a lock conflict and the work has been lost. Two of the solutions that have been proposed are:

- *Multiversion concurrency control protocols*, which we discussed in Section 20.2.6.
- *Advanced transaction models* such as nested transactions, sagas, and multilevel transactions, which we discussed in Section 20.4.

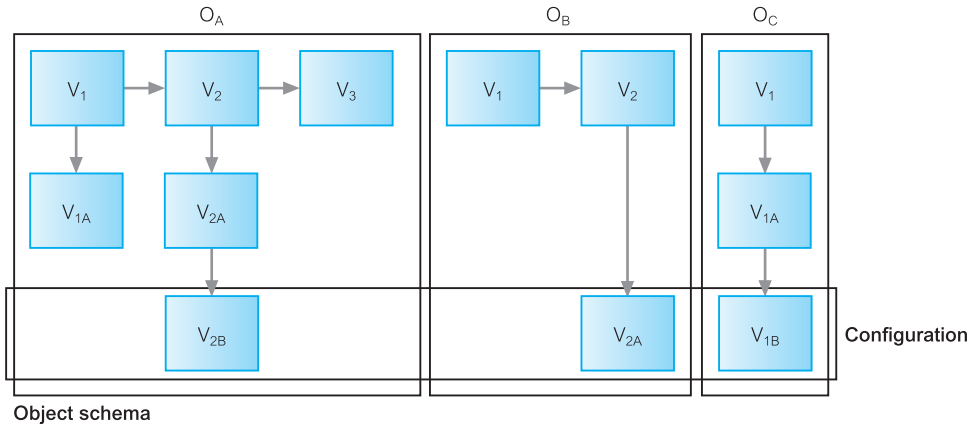


Figure 26.8
Versions and configurations.

26.4.2 Versions

There are many applications that need access to the previous state of an object. For example, the development of a particular design is often an experimental and incremental process, the scope of which changes with time. It is therefore necessary in databases that store designs to keep track of the evolution of design objects and the changes made to a design by various transactions (see for example, Atwood, 1985; Katz *et al.*, 1986; Banerjee *et al.*, 1987a).

The process of maintaining the evolution of objects is known as **version management**. An **object version** represents an identifiable state of an object; a **version history** represents the evolution of an object. Versioning should allow changes to the properties of objects to be managed in such a way that object references always point to the correct version of an object. Figure 26.8 illustrates version management for three objects: O_A, O_B, and O_C. For example, we can determine that object O_A consists of versions V_1, V_2, V_3; V_{1A} is derived from V_1, and V_{2A} and V_{2B} are derived from V_2. This figure also shows an example of a **configuration** of objects, consisting of V_{2B} of O_A, V_{2A} of O_B, and V_{1B} of O_C.

The commercial products Ontos, Versant, ObjectStore, Objectivity/DB, and Itasca provide some form of version management. Itasca identifies three types of version (Kim and Lochovsky, 1989):

- *Transient versions* A transient version is considered unstable and can be updated and deleted. It can be created from new by *checking out* a released version from a *public database* or by deriving it from a working or transient version in a *private database*. In the latter case, the base transient version is promoted to a working version. Transient versions are stored in the creator's private workspace.
- *Working versions* A working version is considered stable and cannot be updated, but it can be deleted by its creator. It is stored in the creator's private workspace.
- *Released versions* A released version is considered stable and cannot be updated or deleted. It is stored in a public database by *checking in* a working version from a private database.

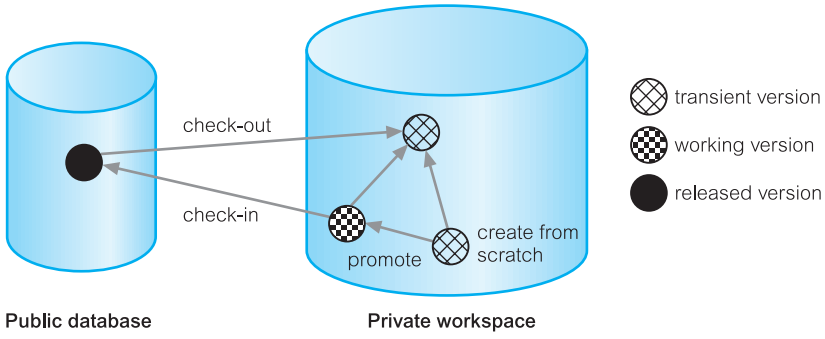


Figure 26.9
Types of versions in Itasca.

These processes are illustrated in Figure 26.9. Owing to the performance and storage overhead in supporting versions, Itasca requires that the application indicate whether a class is **versionable**. When an instance of a versionable class is created, in addition to creating the first version of that instance a **generic object** for that instance is also created, which consists of version management information.

Schema Evolution 26.4.3

Design is an incremental process and evolves with time. To support this process, applications require considerable flexibility in dynamically defining and modifying the database schema. For example, it should be possible to modify class definitions, the inheritance structure, and the specifications of attributes and methods without requiring system shutdown. Schema modification is closely related to the concept of version management discussed above. The issues that arise in schema evolution are complex and not all of them have been investigated in sufficient depth. Typical changes to the schema include (Banerjee *et al.*, 1987b):

(1) Changes to the class definition:
   (a) modifying attributes;
   (b) modifying methods.

(2) Changes to the inheritance hierarchy:
   (a) making a class S the superclass of a class C;
   (b) removing a class S from the list of superclasses of C;
   (c) modifying the order of the superclasses of C.

(3) Changes to the set of classes, such as creating and deleting classes and modifying class names.

The changes proposed to a schema must not leave the schema in an inconsistent state. Itasca and GemStone define rules for schema consistency, called **schema invariants**, which must be complied with as the schema is modified. By way of an example, we consider the schema shown in Figure 26.10. In this figure, inherited attributes and methods are represented by a rectangle. For example, in the Staff class the attributes name and DOB

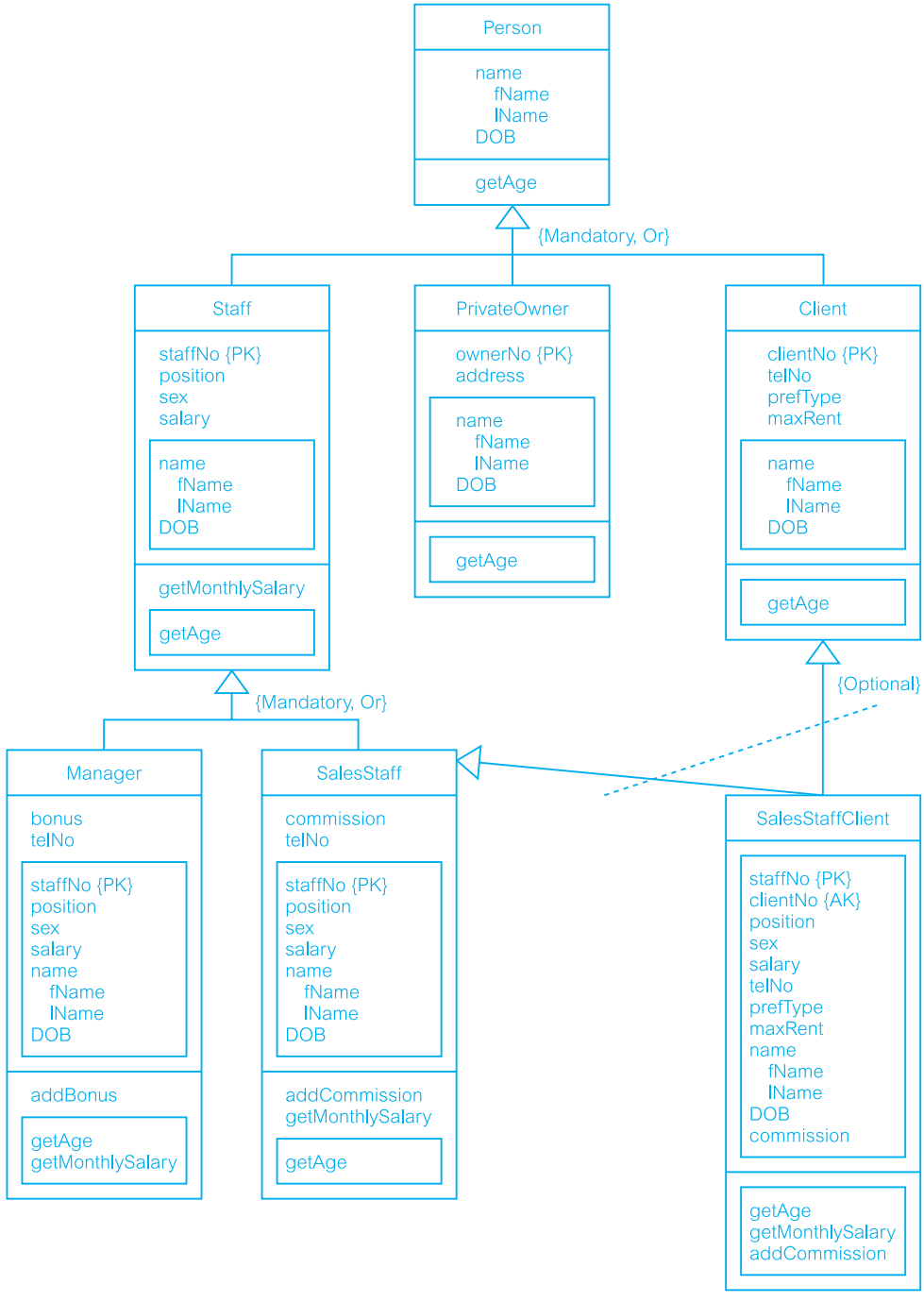


Figure 26.10 Example schema with both single and multiple inheritance.

and the method getAge have been inherited from Person. The rules can be divided into four groups with the following responsibilities:

(1) The resolution of conflicts caused by multiple inheritance and the redefinition of attributes and methods in a subclass.

1.1 *Rule of precedence of subclasses over superclasses*
If an attribute/method of one class is defined with the same name as an attribute/method of a superclass, the definition specified in the subclass takes precedence over the definition of the superclass.

1.2 *Rule of precedence between superclasses of a different origin*
If several superclasses have attributes/methods with the same name but with a different origin, the attribute/method of the first superclass is inherited by the subclass. For example, consider the subclass SalesStaffClient in Figure 26.10, which inherits from SalesStaff and Client. Both these superclasses have an attribute telNo, which is not inherited from a common superclass (which in this case is Person). In this instance, the definition of the telNo attribute in SalesStaffClient is inherited from the first superclass, namely SalesStaff.

1.3 *Rule of precedence between superclasses of the same origin*
If several superclasses have attributes/methods with the same name and the same origin, the attribute/method is inherited only once. If the domain of the attribute has been redefined in any superclass, the attribute with the most specialized domain is inherited by the subclass. If domains cannot be compared, the attribute is inherited from the first superclass. For example, SalesStaffClient inherits name and DOB from both SalesStaff and Client; however, as these attributes are themselves inherited ultimately from Person, they are inherited only once by SalesStaffClient.

(2) The propagation of modifications to subclasses.

2.1 *Rule for propagation of modifications*
Modifications to an attribute/method in a class are always inherited by subclasses, except by those subclasses in which the attribute/method has been redefined. For example, if we deleted the method getAge from Person, this change would be reflected in all subclasses in the entire schema. Note that we could not delete the method getAge directly from a subclass as it is defined in the superclass Person. As another example, if we deleted the method getMonthSalary from Staff, this change would also ripple to Manager, but it would not affect SalesStaff as the method has been redefined in this subclass. If we deleted the attribute telNo from SalesStaff, this version of the attribute telNo would also be deleted from SalesStaffClient but SalesStaffClient would then inherit telNo from Client (see rule 1.2 above).

2.2 *Rule for propagation of modifications in the event of conflicts*
The introduction of a new attribute/method or the modification of the name of an attribute/method is propagated only to those subclasses for which there would be no resulting name conflict.

2.3 *Rule for modification of domains*
The domain of an attribute can only be modified using generalization. The domain of an inherited attribute cannot be made more general than the domain of the original attribute in the superclass.

(3) The aggregation and deletion of inheritance relationships between classes and the creation and removal of classes.

3.1 *Rule for inserting superclasses*
If a class C is added to the list of superclasses of a class C_s, C becomes the last of the superclasses of C_s. Any resulting inheritance conflict is resolved by rules 1.1, 1.2, and 1.3.

3.2 *Rule for removing superclasses*
If a class C has a single superclass C_s, and C_s is deleted from the list of superclasses of C, then C becomes a direct subclass of each direct superclass of C_s. The ordering of the new superclasses of C is the same as that of the superclasses of C_s. For example, if we were to delete the superclass Staff, the subclasses Manager and SalesStaff would then become direct subclasses of Person.

3.3 *Rule for inserting a class into a schema*
If C has no specified superclass, C becomes the subclass of OBJECT (the root of the entire schema).

3.4 *Rule for removing a class from a schema*
To delete a class C from a schema, rule 3.2 is applied successively to remove C from the list of superclasses of all its subclasses. OBJECT cannot be deleted.

(4) Handling of composite objects.
The fourth group relates to those data models that support the concept of composite objects. This group has one rule, which is based on different types of composite object. We omit the detail of this rule and refer the interested reader to the papers by Banerjee *et al.* (1987b) and Kim *et al.* (1989).

26.4.4 Architecture

In this section we discuss two architectural issues: how best to apply the client–server architecture to the OODBMS environment, and the storage of methods.

Client–server

Many commercial OODBMSs are based on the client–server architecture to provide data to users, applications, and tools in a distributed environment (see Section 2.6). However, not all systems use the same client–server model. We can distinguish three basic architectures for a client–server DBMS that vary in the functionality assigned to each component (Loomis, 1992), as depicted in Figure 26.11:

- *Object server* This approach attempts to distribute the processing between the two components. Typically, the server process is responsible for managing storage, locks, commits to secondary storage, logging and recovery, enforcing security and integrity, query optimization, and executing stored procedures. The client is responsible for transaction management and interfacing to the programming language. This is the best architecture for cooperative, object-to-object processing in an open, distributed environment.

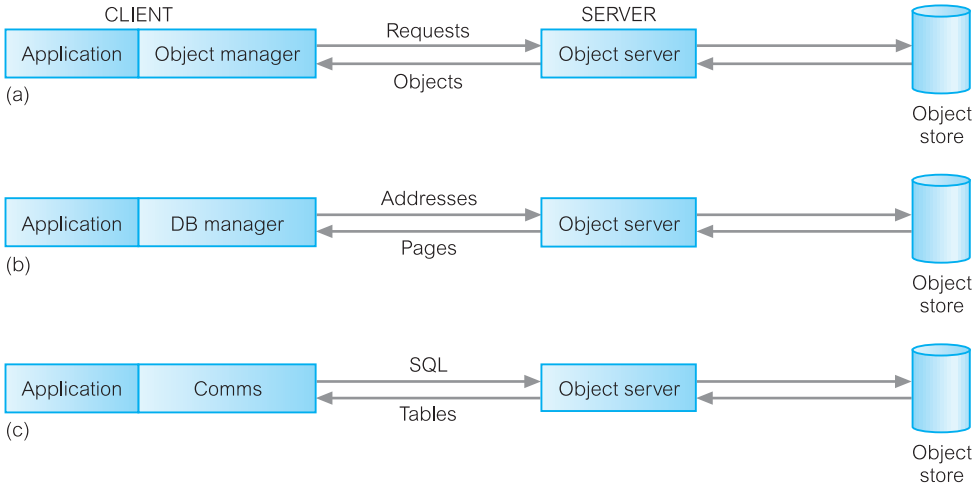


Figure 26.11
Client–server architectures: (a) object server; (b) page server; (c) database server.

- *Page server* In this approach, most of the database processing is performed by the client. The server is responsible for secondary storage and providing pages at the client's request.
- *Database server* In this approach, most of the database processing is performed by the server. The client simply passes requests to the server, receives results, and passes them on to the application. This is the approach taken by many RDBMSs.

In each case, the server resides on the same machine as the physical database; the client may reside on the same or different machine. If the client needs access to databases distributed across multiple machines, then the client communicates with a server on each machine. There may also be a number of clients communicating with one server, for example, one client for each user or application.

Storing and executing methods

There are two approaches to handling methods: store the methods in external files, as shown in Figure 26.12(a), and store the methods in the database, as shown in Figure 26.12(b). The first approach is similar to function libraries or Application Programming Interfaces (APIs) found in traditional DBMSs, in which an application program interacts with a DBMS by linking in functions supplied by the DBMS vendor. With the second approach, methods are stored in the database and are dynamically bound to the application at runtime. The second approach offers several benefits:

- *It eliminates redundant code* Instead of placing a copy of a method that accesses a data element in every program that deals with that data, the method is stored only once in the database.
- *It simplifies modifications* Changing a method requires changing it in one place only. All the programs automatically use the updated method. Depending on the nature of the change, rebuilding, testing, and redistribution of programs may be eliminated.

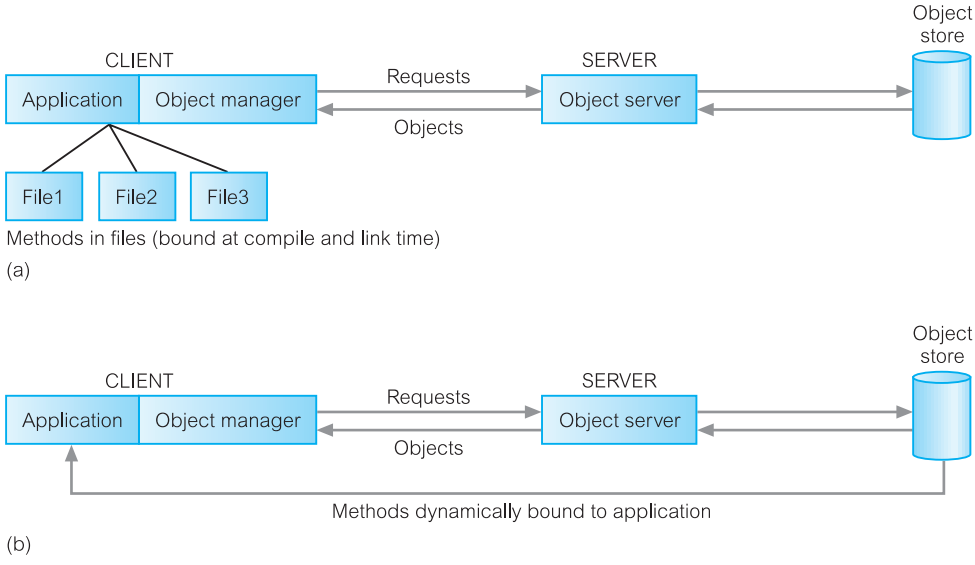


Figure 26.12
Strategies for handling methods: (a) storing methods outside database; (b) storing methods in database.

- *Methods are more secure* Storing the methods in the database gives them all the benefits of security provided automatically by the OODBMS.
- *Methods can be shared concurrently* Again, concurrent access is provided automatically by the OODBMS. This also prevents multiple users making different changes to a method simultaneously.
- *Improved integrity* Storing the methods in the database means that integrity constraints can be enforced consistently by the OODBMS across all applications.

The products GemStone and Itasca allow methods to be stored and activated from within the database.

26.4.5 Benchmarking

Over the years, various database benchmarks have been developed as a tool for comparing the performance of DBMSs and are frequently referred to in academic, technical, and commercial literature. Before we examine two object-oriented benchmarks, we first provide some background to the discussion. Complete descriptions of these benchmarks are outwith the scope of this book but for full details of the benchmarks the interested reader is referred to Gray (1993).

Wisconsin benchmark

Perhaps the earliest DBMS benchmark was the Wisconsin benchmark, which was developed to allow comparison of particular DBMS features (Bitton *et al.*, 1983). It consists of a set of tests as a single user covering:

- updates and deletes involving both key and non-key attributes;
- projections involving different degrees of duplication in the attributes and selections with different selectivities on indexed, non-index, and clustered attributes;
- joins with different selectivities;
- aggregate functions.

The original Wisconsin benchmark was based on three relations: one relation called Onektup with 1000 tuples, and two others called Tenktup1/Tenktup2 with 10,000 tuples each. This benchmark has been generally useful although it does not cater for highly skewed attribute distributions and the join queries used are relatively simplistic.

Owing to the importance of accurate benchmarking information, a consortium of manufacturers formed the **Transaction Processing Council** (TPC) in 1988 to formulate a series of transaction-based test suites to measure database/TP environments. Each consists of a printed specification and is accompanied by ANSI 'C' source code, which populates a database with data according to a preset standardized structure.

TPC-A and TPC-B benchmarks

TPC-A and TPC-B are based on a simple banking transaction. TPC-A measures online transaction processing (OLTP) performance covering the time taken by the database server, network, and any other components of the system but excluding user interaction. TPC-B measures only the performance of the database server. A transaction simulates the transfer of money to or from an account with the following actions:

- update the account record (Account relation has 100,000 tuples);
- update the teller record (Teller relation has 10 tuples);
- update the branch record (Branch relation has 1 tuple);
- update a history record (History relation has 2,592,000 tuples);
- return the account balance.

The cardinalities quoted above are for a minimal configuration but the database can be scaled in multiples of this configuration. As these actions are performed on single tuples, important aspects of the system are not measured (for example, query planning and join execution).

TPC-C benchmark

TPC-A and TPC-B are obsolescent and are being replaced by TPC-C, which is based on an order entry application. The underlying database schema and the range of queries are more complex than TPC-A, thereby providing a much more comprehensive test of a DBMS's performance. There are five transactions defined covering a new order, a payment, an order status inquiry, a delivery, and a stock level inquiry.

Other benchmarks

The Transaction Processing Council has defined a number of other benchmarks, such as:

- TPC-H, for *ad hoc*, decision support environments where users do not know which queries will be executed;
- TPC-R, for business reporting within decision support environments where users run a standard set of queries against a database system;
- TPC-W, a transactional Web benchmark for e-Commerce, where the workload is performed in a controlled Internet commerce environment that simulates the activities of a business-oriented transactional Web server.

The Transaction Processing Council publishes the results of the benchmarks on its Web site (www.tpc.org).

OO1 benchmark

The Object Operations Version 1 (OO1) benchmark is intended as a generic measure of OODBMS performance (Cattell and Skeen, 1992). It was designed to reproduce operations that are common in the advanced engineering applications discussed in Section 25.1, such as finding all parts connected to a random part, all parts connected to one of those parts, and so on, to a depth of seven levels. The benchmark involves:

- random retrieval of 1000 parts based on the primary key (the part number);
- random insertion of 100 new parts and 300 randomly selected connections to these new parts, committed as one transaction;
- random parts explosion up to seven levels deep, retrieving up to 3280 parts.

In 1989 and 1990, the OO1 benchmark was run on the OODBMSs GemStone, Ontos, ObjectStore, Objectivity/DB, and Versant, and the RDBMSs INGRES and Sybase. The results showed an average 30-fold performance improvement for the OODBMSs over the RDBMSs. The main criticism of this benchmark is that objects are connected in such a way as to prevent clustering (the closure of any object is the entire database). Thus, systems that have good navigational access at the expense of any other operations perform well against this benchmark.

OO7 benchmark

In 1993, the University of Wisconsin released the OO7 benchmark, based on a more comprehensive set of tests and a more complex database. OO7 was designed for detailed comparisons of OODBMS products (Carey *et al*., 1993). It simulates a CAD/CAM environment and tests system performance in the area of object-to-object navigation over cached data, disk-resident data, and both sparse and dense traversals. It also tests indexed and non-indexed updates of objects, repeated updates, and the creation and deletion of objects.

The OO7 database schema is based on a complex parts hierarchy, where each part has associated documentation, and modules (objects at the top level of the hierarchy) have a manual. The tests are split into two groups. The first group is designed to test:

- traversal speed (simple test of navigational performance similar to that measured in OO1);

- traversal with updates (similar to the first test, but with updates covering every atomic part visited, a part in every composite part, every part in a composite part four times);
- operations on the documentation.

The second group contains declarative queries covering exact match, range searches, path lookup, scan, a simulation of the make utility, and join. To facilitate its use, a number of sample implementations are available via anonymous ftp from ftp.cs.wisc.edu.

26.5 Advantages and Disadvantages of OODBMSs

OODBMSs can provide appropriate solutions for many types of advanced database applications. However, there are also disadvantages. In this section we examine these advantages and disadvantages.

26.5.1 Advantages

The advantages of OODBMSs are listed in Table 26.2.

Enriched modeling capabilities

The object-oriented data model allows the 'real world' to be modeled more closely. The object, which encapsulates both state and behavior, is a more natural and realistic representation of real-world objects. An object can store all the relationships it has with other objects, including many-to-many relationships, and objects can be formed into complex objects that the traditional data models cannot cope with easily.

Extensibility

OODBMSs allow new data types to be built from existing types. The ability to factor out common properties of several classes and form them into a superclass that can be shared

Table 26.2 Advantages of OODBMSs.

Enriched modeling capabilities
Extensibility
Removal of impedance mismatch
More expressive query language
Support for schema evolution
Support for long-duration transactions
Applicability to advanced database applications
Improved performance

with subclasses can greatly reduce redundancy within systems and, as we stated at the start of this chapter, is regarded as one of the main advantages of object orientation. Overriding is an important feature of inheritance as it allows special cases to be handled easily, with minimal impact on the rest of the system. Further, the reusability of classes promotes faster development and easier maintenance of the database and its applications.

It is worthwhile pointing out that if domains were properly implemented, RDBMSs would be able to provide the same functionality as OODBMSs are claimed to have. A domain can be perceived as a data type of arbitrary complexity with scalar values that are encapsulated, and that can be operated on only by predefined functions. Therefore, an attribute defined on a domain in the relational model can contain anything, for example, drawings, documents, images, arrays, and so on (Date, 2000). In this respect, domains and object classes are arguably the same thing. We return to this point in Section 28.2.2.

Removal of impedance mismatch

A single language interface between the Data Manipulation Language (DML) and the programming language overcomes the impedance mismatch. This eliminates many of the inefficiencies that occur in mapping a declarative language such as SQL to an imperative language such as 'C'. We also find that most OODBMSs provide a DML that is computationally complete compared with SQL, the standard language for RDBMSs.

More expressive query language

Navigational access from one object to the next is the most common form of data access in an OODBMS. This is in contrast to the associative access of SQL (that is, declarative statements with selection based on one or more predicates). Navigational access is more suitable for handling parts explosion, recursive queries, and so on. However, it is argued that most OODBMSs are tied to a particular programming language that, although convenient for programmers, is not generally usable by end-users who require a declarative language. In recognition of this, the ODMG standard specifies a declarative query language based on an object-oriented form of SQL (see Section 27.2.4).

Support for schema evolution

The tight coupling between data and applications in an OODBMS makes schema evolution more feasible. Generalization and inheritance allow the schema to be better structured, to be more intuitive, and to capture more of the semantics of the application.

Support for long-duration transactions

Current relational DBMSs enforce serializability on concurrent transactions to maintain database consistency (see Section 20.2.2). Some OODBMSs use a different protocol to handle the types of long-duration transaction that are common in many advanced database applications. This is an arguable advantage: as we have already mentioned in Section 25.2, there is no structural reason why such transactions cannot be provided by an RDBMS.

Applicability to advanced database applications

As we discussed in Section 25.1, there are many areas where traditional DBMSs have not been particularly successful, such as, computer-aided design (CAD), computer-aided software engineering (CASE), office information systems (OISs), and multimedia systems. The enriched modeling capabilities of OODBMSs have made them suitable for these applications.

Improved performance

As we mentioned in Section 26.4.5, there have been a number of benchmarks that have suggested OODBMSs provide significant performance improvements over relational DBMSs. For example, in 1989 and 1990, the OO1 benchmark was run on the OODBMSs GemStone, Ontos, ObjectStore, Objectivity/DB, and Versant, and the RDBMSs INGRES and Sybase. The results showed an average 30-fold performance improvement for the OODBMS over the RDBMS, although it has been argued that this difference in performance can be attributed to architecture-based differences, as opposed to model-based differences. However, dynamic binding and garbage collection in OODBMSs may compromise this performance improvement.

It has also been argued that these benchmarks target engineering applications, which are more suited to object-oriented systems. In contrast, it has been suggested that RDBMSs outperform OODBMSs with traditional database applications, such as online transaction processing (OLTP).

Disadvantages 26.5.2

The disadvantages of OODBMSs are listed in Table 26.3.

Lack of universal data model

As we discussed in Section 26.1, there is no universally agreed data model for an OODBMS, and most models lack a theoretical foundation. This disadvantage is seen as a

Table 26.3 Disadvantages of OODBMSs.

Lack of universal data model
Lack of experience
Lack of standards
Competition
Query optimization compromises encapsulation
Locking at object level may impact performance
Complexity
Lack of support for views
Lack of support for security

significant drawback and is comparable to pre-relational systems. However, the ODMG proposed an object model that has become the *de facto* standard for OODBMSs. We discuss the ODMG object model in Section 27.2.

Lack of experience

In comparison to RDBMSs, the use of OODBMSs is still relatively limited. This means that we do not yet have the level of experience that we have with traditional systems. OODBMSs are still very much geared towards the programmer, rather than the naïve end-user. Furthermore, the learning curve for the design and management of OODBMSs may be steep, resulting in resistance to the acceptance of the technology. While the OODBMS is limited to a small niche market, this problem will continue to exist.

Lack of standards

There is a general lack of standards for OODBMSs. We have already mentioned that there is no universally agreed data model. Similarly, there is no standard object-oriented query language. Again, the ODMG specified an Object Query Language (OQL) that has become a *de facto* standard, at least in the short term (see Section 27.2.4). This lack of standards may be the single most damaging factor for the adoption of OODBMSs.

Competition

Perhaps one of the most significant issues that face OODBMS vendors is the competition posed by the RDBMS and the emerging ORDBMS products. These products have an established user base with significant experience available, SQL is an approved standard and ODBC is a *de facto* standard, the relational data model has a solid theoretical foundation, and relational products have many supporting tools to help both end-users and developers.

Query optimization compromises encapsulation

Query optimization requires an understanding of the underlying implementation to access the database efficiently. However, this compromises the concept of encapsulation. The OODBMS Manifesto, discussed in Section 26.1.4, suggests that this may be acceptable although, as we discussed, this seems questionable.

Locking at object level may impact performance

Many OODBMSs use locking as the basis for a concurrency control protocol. However, if locking is applied at the object level, locking of an inheritance hierarchy may be problematic, as well as impacting performance. We examined how to lock hierarchies in Section 20.2.8.

Complexity

The increased functionality provided by an OODBMS, such as the illusion of a single-level storage model, pointer swizzling, long-duration transactions, version management, and

schema evolution, is inherently more complex than that of traditional DBMSs. In general, complexity leads to products that are more expensive to buy and more difficult to use.

Lack of support for views

Currently, most OODBMSs do not provide a view mechanism, which, as we have seen previously, provides many advantages such as data independence, security, reduced complexity, and customization (see Section 6.4).

Lack of support for security

Currently, OODBMSs do not provide adequate security mechanisms. Most mechanisms are based on a coarse granularity, and the user cannot grant access rights on individual objects or classes. If OODBMSs are to expand fully into the business field, this deficiency must be rectified.

Chapter Summary

- An **OODBMS** is a manager of an OODB. An **OODB** is a persistent and sharable repository of objects defined in an OODM. An **OODM** is a data model that captures the semantics of objects supported in object-oriented programming. There is no universally agreed OODM.
- The functional data model (FDM) shares certain ideas with the object approach including object identity, inheritance, overloading, and navigational access. In the FDM, any data retrieval task can be viewed as the process of evaluating and returning the result of a function with zero, one, or more arguments. In the FDM, the main modeling primitives are **entities** (either **entity types** or **printable entity types**) and **functional relationships**.
- A **persistent programming language** is a language that provides its users with the ability to (transparently) preserve data across successive executions of a program. Data in a persistent programming language is independent of any program, able to exist beyond the execution and lifetime of the code that created it. However, such languages were originally intended to provide neither full database functionality nor access to data from multiple languages.
- The *Object-Oriented Database System Manifesto* proposed the following mandatory object-oriented characteristics: complex objects, object identity, encapsulation, types/classes, inheritance, dynamic binding, a computationally complete DML, and extensible data types.
- Alternative approaches for developing an OODBMS include: extend an existing object-oriented programming language with database capabilities; provide extensible OODBMS libraries; embed OODB language constructs in a conventional host language; extend an existing database language with object-oriented capabilities; and develop a novel database data model/data language.
- Perhaps two of the most important concerns from the programmer's perspective are performance and ease of use. Both are achieved by having a more seamless integration between the programming language and the DBMS than that provided with traditional database systems. Conventional DBMSs have a two-level storage model: the application storage model in main or virtual memory, and the database storage model on disk. In contrast, an OODBMS tries to give the illusion of a single-level storage model, with a similar representation in both memory and in the database stored on disk.

- There are two types of **OID**: logical OIDs that are independent of the physical location of the object on disk, and physical OIDs that encode the location. In the former case, a level of indirection is required to look up the physical address of the object on disk. In both cases, however, an OID is different in size from a standard in-memory pointer, which need only be large enough to address all virtual memory.
- To achieve the required performance, an OODBMS must be able to convert OIDs to and from in-memory pointers. This conversion technique has become known as **pointer swizzling** or **object faulting**, and the approaches used to implement it have become varied, ranging from software-based residency checks to page-faulting schemes used by the underlying hardware.
- Persistence schemes include checkpointing, serialization, explicit paging, and orthogonal persistence. **Orthogonal persistence** is based on three fundamental principles: persistence independence, data type orthogonality, and transitive persistence.
- Advantages of OODBMSs include enriched modeling capabilities, extensibility, removal of impedance mismatch, more expressive query language, support for schema evolution and long-duration transactions, applicability to advanced database applications, and performance. Disadvantages include lack of universal data model, lack of experience, lack of standards, query optimization compromises encapsulation, locking at the object level impacts performance, complexity, and lack of support for views and security.

Review Questions

26.1 Compare and contrast the different definitions of object-oriented data models.

26.2 Describe the main modeling component of the functional data model.

26.3 What is a persistent programming language and how does it differ from an OODBMS?

26.4 Discuss the difference between the two-level storage model used by conventional DBMSs and the single-level storage model used by OODBMSs.

26.5 How does this single-level storage model affect data access?

26.6 Describe the main strategies that can be used to create persistent objects.

26.7 What is pointer swizzling? Describe the different approaches to pointer swizzling.

26.8 Describe the types of transaction protocol that can be useful in design applications.

26.9 Discuss why version management may be a useful facility for some applications.

26.10 Discuss why schema control may be a useful facility for some applications.

26.11 Describe the different architectures for an OODBMS.

26.12 List the advantages and disadvantages of an OODBMS.

Exercises

26.13 You have been asked by the Managing Director of *DreamHome* to investigate and prepare a report on the applicability of an OODBMS for the organization. The report should compare the technology of the RDBMS with that of the OODBMS, and should address the advantages and disadvantages of implementing an OODBMS within the organization, and any perceived problem areas. Finally, the report should contain a fully justified set of conclusions on the applicability of the OODBMS for *DreamHome*.

26.14 For the relational Hotel schema in the Exercises at the end of Chapter 3, suggest a number of methods that may be applicable to the system. Produce an object-oriented schema for the system.

26.15 Produce an object-oriented database design for the *DreamHome* case study documented in Appendix A. State any assumptions necessary to support your design.

26.16 Produce an object-oriented database design for the *University Accommodation Office* case study presented in Appendix B.1. State any assumptions necessary to support your design.

26.17 Produce an object-oriented database design for the *EasyDrive School of Motoring* case study presented in Appendix B.2. State any assumptions necessary to support your design.

26.18 Produce an object-oriented database design for the *Wellmeadows Hospital* case study presented in Appendix B.3. State any assumptions necessary to support your design.

26.19 Repeat Exercises 26.14 to 26.18 but produce a schema using the functional data model. Diagrammatically illustrate each schema.

26.20 Using the rules for schema consistency given in Section 26.4.3 and the sample schema given in Figure 26.10, consider each of the following modifications and state what the effect of the change should be to the schema:

(a) adding an attribute to a class;
(b) deleting an attribute from a class;
(c) renaming an attribute;
(d) making a class S a superclass of a class C;
(e) removing a class S from the list of superclasses of a class C;
(f) creating a new class C;
(g) deleting a class;
(h) modifying class names.

Chapter

27 Object-Oriented DBMSs – Standards and Systems

Chapter Objectives

In this chapter you will learn:

- About the Object Management Group (OMG) and the Object Management Architecture (OMA).
- The main features of the Common Object Request Broker Architecture (CORBA).
- The main features of the other OMG standards including UML, MOF, XMI, CWM, and the Model-Driven Architecture (MDA).
- The main features of the new Object Data Management Group (ODMG) Object Data Standard:
  - Object Model;
  - Object Definition Language (ODL);
  - Object Query Language (OQL);
  - Object Interchange Format (OIF);
  - language bindings.
- The main features of ObjectStore, a commercial OODBMS:
  - the ObjectStore architecture;
  - data definition in ObjectStore;
  - data manipulation in ObjectStore.

In the previous chapter we examined some of the issues associated with Object-Oriented Database Management Systems (OODBMSs). In this chapter we continue our study of these systems and examine the object model and specification languages proposed by the Object Data Management Group (ODMG). The ODMG object model is important because it specifies a standard model for the semantics of database objects and supports interoperability between compliant systems. It has become the *de facto* standard for OODBMSs. To put the discussion of OODBMSs into a commercial context, we also examine the architecture and functionality of ObjectStore, a commercial OODBMS.

Structure of this Chapter

As the ODMG model is a superset of the model supported by Object Management Group (OMG), we provide an overview of the OMG and the OMG architecture in Section 27.1. In Section 27.2 we discuss the ODMG object model and ODMG specification languages. Finally, in Section 27.3, to illustrate the architecture and functionality of commercial OODBMSs, we examine one such system in detail, namely ObjectStore.

In order to benefit fully from this chapter, the reader needs to be familiar with the contents of Chapters 25 and 26. The examples in this chapter are once again drawn from the *DreamHome* case study documented in Section 10.4 and Appendix A.

27.1 Object Management Group

To put the ODMG object model into perspective, we start with a brief presentation of the function of the Object Management Group and the architecture and some of the specification languages that it has proposed.

27.1.1 Background

The OMG is an international non-profit-making industry consortium founded in 1989 to address the issues of object standards. The group has more than 400 member organizations including virtually all platform vendors and major software vendors such as Sun Microsystems, Borland, AT&T/NCR, HP, Hitachi, Computer Associates, Unisys, and Oracle. All these companies have agreed to work together to create a set of standards acceptable to all. The primary aims of the OMG are promotion of the object-oriented approach to software engineering and the development of standards in which the location, environment, language, and other characteristics of objects are completely transparent to other objects.

The OMG is not a recognized standards group, unlike the International Organization for Standardization (ISO) or national bodies such as the American National Standards Institute (ANSI) or the Institute of Electrical and Electronics Engineers (IEEE). The aim of the OMG is to develop *de facto* standards that will eventually be acceptable to ISO/ANSI. The OMG does not actually develop or distribute products, but will certify compliance with the OMG standards.

In 1990, the OMG first published its Object Management Architecture (OMA) Guide document and it has gone through a number of revisions since then (Soley, 1990, 1992, 1995). This guide specifies a single terminology for object-oriented languages, systems, databases, and application frameworks; an abstract framework for object-oriented systems; a set of technical and architectural goals; and a reference model for distributed applications using object-oriented techniques. Four areas of standardization were identified for the reference model: the Object Model (OM), the Object Request Broker (ORB), the Object Services, and the Common Facilities, as illustrated in Figure 27.1.

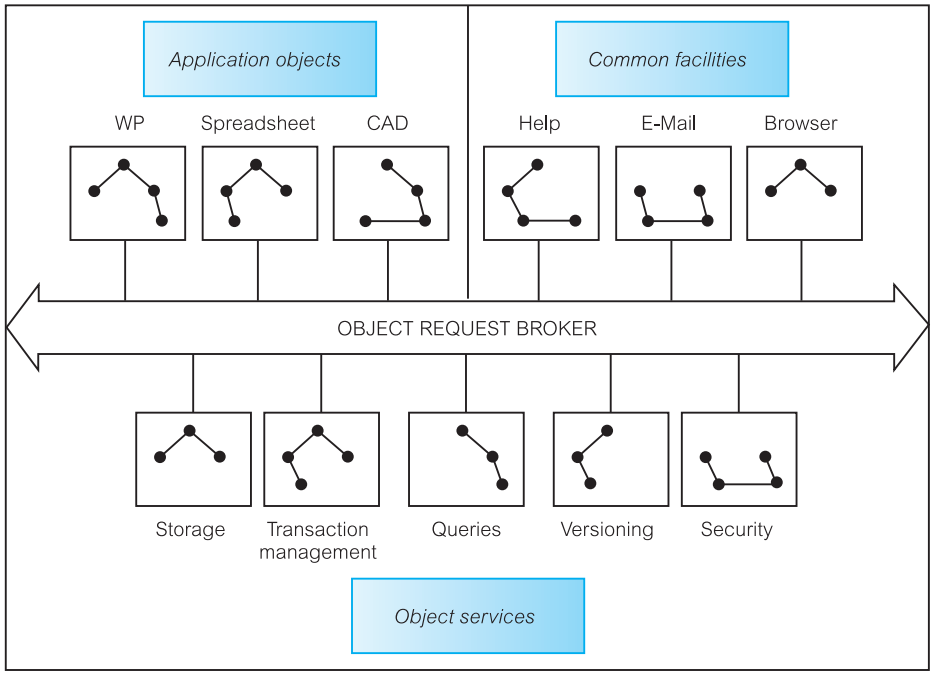


Figure 27.1
Object reference model.

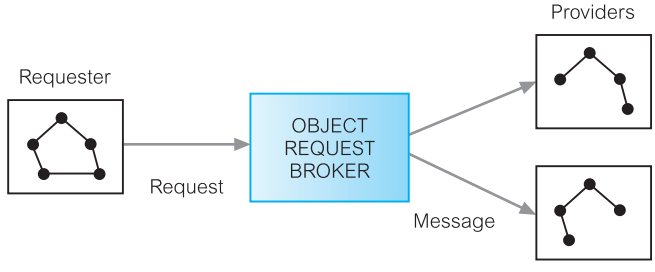


Figure 27.2
OMG Object Model.

The Object Model

The OM is a design-portable abstract model for communicating with OMG-compliant object-oriented systems (see Figure 27.2). A requester sends a request for object services to the ORB, which keeps track of all the objects in the system and the types of service they can provide. The ORB then forwards the message to a provider who acts on the message and passes a response back to the requester via the ORB. As we shall see shortly, the OMG OM is a subset of the ODMG OM.

The Object Request Broker

The ORB handles distribution of messages between application objects in a highly interoperable manner. In effect, the ORB is a distributed 'software bus' (or telephone exchange)

that enables objects (requesters) to make and receive requests and responses from a provider. On receipt of a response from the provider, the ORB translates the response into a form that the original requester can understand. The ORB is analogous to the X500 electronic mail communications standard, wherein a requester can issue a request to another application or node without having detailed knowledge of its directory services structure. In this way, the ORB removes much of the need for complex Remote Procedure Calls (RPCs) by providing the mechanisms by which objects make and receive requests and responses transparently. The objective is to provide interoperability between applications in a heterogeneous distributed environment and to connect multiple object systems transparently.

The Object Services

The Object Services provide the main functions for realizing basic object functionality. Many of these services are database-oriented as listed in Table 27.1.

The Common Facilities

The Common Facilities comprise a set of tasks that many applications must perform but are traditionally duplicated within each one, such as printing and electronic mail facilities. In the OMG Reference Model they are made available through OMA-compliant class interfaces. In the object references model, the common facilities are split into *horizontal common facilities* and *vertical domain facilities*. There are currently only four common facilities: Printing, Secure Time, Internationalization, and the Mobile Agent. Domain facilities are specific interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecommunications, e-Commerce, and Transportation.

The Common Object Request Broker Architecture 27.1.2

The Common Object Request Broker Architecture (CORBA) defines the architecture of ORB-based environments. This architecture is the basis of any OMG component, defining the parts that form the ORB and its associated structures. Using the communication protocols GIOP (General Inter-Object Protocol) or IIOP (Internet Inter-Object Protocol, which is GIOP built on top of TCP/IP), a CORBA-based program can interoperate with another CORBA-based program across a variety of vendors, platforms, operating systems, programming languages, and networks.

CORBA 1.1 was introduced in 1991 and defined an Interface Definition Language and Application Programming Interfaces that enable client–server interaction with a specific implementation of an ORB. CORBA 2.0 was released in December 1994 and provided improved interoperability by specifying how ORBs from different vendors can interoperate. CORBA 2.1 was released during the latter part of 1997, CORBA 2.2 in 1998, and CORBA 2.3 in 1999 (OMG, 1999). Some of the elements of CORBA are:

- An implementation-neutral **Interface Definition Language** (IDL), which permits the description of class interfaces independent of any particular DBMS or programming

Table 27.1 OMG Object Services.

Object service	Description
Collection	Provides a uniform way to create and manipulate most common collections generically. Examples are sets, bags, queues, stacks, lists, and binary trees.
Concurrency control	Provides a lock manager that enables multiple clients to coordinate their access to shared resources.
Event management	Allows components to dynamically register or unregister their interest in specific events.
Externalization	Provides protocols and conventions for externalizing and internalizing objects. **Externalization** records the state of an object as a stream of data (for example, in memory, on disk, across networks), and then **internalization** creates a new object from it in the same or different process.
Licensing	Provides operations for metering the use of components to ensure fair compensation for their use, and protect intellectual property.
Lifecycle	Provides operations for creating, copying, moving, and deleting groups of related objects.
Naming	Provides facilities to bind a name to an object relative to a naming context.
Persistence	Provides interfaces to the mechanisms for storing and managing objects persistently.
Properties	Provides operations to associate named values (properties) with any (external) component.
Query	Provides declarative query statements with predicates and includes the ability to invoke operations and to invoke other object services.
Relationship	Provides a way to create dynamic associations between components that know nothing of each other.
Security	Provides services such as identification and authentication, authorization and access control, auditing, security of communication, non-repudiation, and administration.
Time	Maintains a single notion of time across different machines.
Trader	Provides a matchmaking service for objects. It allows objects to dynamically advertise their services and other objects to register for a service.
Transactions	Provides two-phase commit (2PC) coordination among recoverable components using either flat or nested transactions.

language. There is an IDL compiler for each supported programming language, allowing programmers to use constructs that they are familiar with.

- A **type model** that defines the values that can be passed over the network.
- An **Interface Repository** that stores persistent IDL definitions. The Interface Repository can be queried by a client application to obtain a description of all the registered object interfaces, the methods they support, the parameters they require, and the exceptions that may arise.

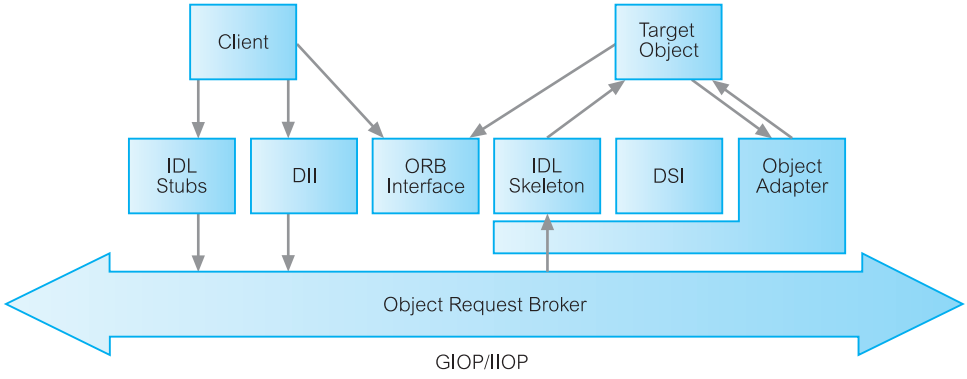


Figure 27.3
The CORBA ORB architecture.

- Methods for getting the interfaces and specifications of objects.
- Methods for transforming OIDs to and from strings.

As illustrated in Figure 27.3, CORBA provides two mechanisms for clients to issue requests to objects:

- static invocations using interface-specific stubs and skeletons;
- dynamic invocations using the Dynamic Invocation Interface.

Static method invocation

From the IDL definitions, CORBA objects can be mapped into particular programming languages or object systems, such as 'C', C++, Smalltalk, and Java. An IDL compiler generates three files:

- a header file, which is included in both the client and server;
- a client source file, which contains interface **stubs** that are used to transmit the requests to the server for the interfaces defined in the compiled IDL file;
- a server source file, which contains **skeletons** that are completed on the server to provide the required behavior.

Dynamic method invocation

Static method invocation requires the client to have an IDL stub for each interface it uses on the server. Clearly, this prevents the client from using the service of a newly created object if it does not know its interface, and therefore does not have the corresponding stubs to generate the request. To overcome this, the **Dynamic Invocation Interface** (DII) allows the client to identify objects and their interfaces at runtime, and then to construct and invoke these interfaces and receive the results of these dynamic invocations. The specifications of the objects and the services they provide are stored in the Interface Repository.

A server-side analog of DII is the **Dynamic Skeleton Interface** (DSI), which is a way to deliver requests from the ORB to an object implementation that does not have compile-time knowledge of the object it is implementing. With DSI the operation is no

longer accessed through an operation-specific skeleton generated from an IDL interface specification, but instead it is reached through an interface that provides access to the operation name and parameters using information from the Interface Repository.

Object Adapter

Also built into the architecture is the Object Adapter, which is the main way a (server-side) object implementation accesses services provided by the ORB. An Object Adapter is responsible for the registration of object implementations, generation and interpretation of object references, static and dynamic method invocation, object and implementation activation and deactivation, and security coordination. CORBA requires a standard adapter known as the Basic Object Adapter.

In 1999, the OMG announced release 3 of CORBA, which adds firewall standards for communicating over the Internet, quality of service parameters, and CORBAcomponents. CORBAcomponents allow programmers to activate fundamental services at a higher level and was intended as a vendor-neutral, language-independent, component middleware, but the OMG has now embraced Enterprise JavaBeans (EJB), a middle-tier specification that allows only Java as a programming language in the middle tier (see Section 29.9). In the literature, CORBA 2 generally refers to CORBA interoperability and the IIOP protocol, and CORBA 3 refers to the CORBA Component Model. There are many vendors of CORBA ORBs on the market, with IONA's Orbix and Inprise's Visibroker being popular examples.

27.1.3 Other OMG Specifications

The OMG has also developed a number of specifications for modeling distributed software architectures and systems along with their CORBA interfaces. There are four complementary specifications currently available:

(1) *Unified Modeling Language (UML)* provides a common language for describing software models. It is commonly defined as 'a standard language for specifying, constructing, visualizing, and documenting the artifacts of a software system'. We used the class diagram notation of the UML as the basis for the ER models we created in Part 4 of this book and we discussed the other components of the UML in Section 25.7.

(2) *Meta-Object Facility (MOF)* defines a common, abstract language for the specification of *metamodels*. In the MOF context, a model is a collection of related metadata, and metadata that describes metadata is called meta-metadata, and a model that consists of meta-metadata is called a metamodel. In other words, MOF is a meta-metamodel or model of a metamodel (sometimes called an *ontology*). For example, the UML supports a number of different diagrams such as class diagrams, use case diagrams, and activity diagrams. Each of these diagram types is a different type of metamodel. MOF also defines a framework for implementing repositories that hold metadata described by the metamodels. The framework provides mappings to transform MOF metamodels into metadata APIs. Thus, MOF enables dissimilar metamodels that represent different domains to be used in an interoperable way. CORBA, UML, and CWM (see below) are all MOF-compliant metamodels.

Table 27.2 OMG metadata architecture.

Meta-level	MOF terms	Examples
M3	meta-metamodel	The 'MOF Model'
M2	metamodel, meta-metadata	UML metamodel CWM metamodel
M1	model, metadata	UML models CWM metadata
M0	object, data	Modeled systems Warehouse data

The MOF metadata framework is typically depicted as a four layer architecture as shown in Table 27.2. MOF is important for the UML to ensure that each UML model type is defined in a consistent way. For example, MOF ensures that a 'class' in a class diagram has an exact relationship to a 'use case' in a use case diagram or an 'activity' in an activity diagram.

(3) *XML Metadata Interchange (XMI)* maps the MOF to XML. XMI defines how XML tags are used to represent MOF-compliant models in XML. An MOF-based metamodel can be translated to a Document Type Definition (DTD) or an XML Schema and a model is translated to an XML document that is consistent with its DTD or XML Schema. XMI is intended to be a 'stream' format, so that it can either be stored in a traditional file system or be *streamed* across the Internet from a database or repository. We discuss XML, DTDs, and XML Schema in Chapter 30.

(4) *Common Warehouse Metamodel (CWM)* defines a metamodel representing both the business and technical metadata that is commonly found in data warehousing and business intelligence domains. The OMG recognized that metadata management and integration are significant challenges in these fields, where products have their own definition and format for metadata. CWM standardizes how to represent database models (schemas), schema transformation models, OLAP and data mining models. It is used as the basis for interchanging instances of metadata between heterogeneous, multi-vendor software systems. CWM is defined in terms of MOF with the UML as the modeling notation (and the base metamodel) and XMI is the interchange mechanism.

As indicated in Figure 27.4, CWM consists of a number of sub-metamodels organized into 18 packages that represent common warehouse metadata:

(a) *data resource metamodels* support the ability to model legacy and non-legacy data resources including object-oriented, relational, record, multi-dimensional, and XML data resources (Figure 27.5 shows the CWM Relational Data Metamodel);

(b) *data analysis metamodels* represent such things as data transformations, OLAP (OnLine Analytical Processing), data mining, and information visualization;

(c) *warehouse management metamodels* represent standard warehouse processes and the results of warehouse operations;

(d) *foundation metamodel* supports the specification of various general services such as data types, indexes, and component-based software deployment.

Figure 27.4
CWM layers and package structure.

Management	Warehouse process			Warehouse operation		
Analysis	Transformation		OLAP	Data mining	Information visualization	Business nomenclature
Resource	Object-oriented (UML)	Relational	Record	Multi-dimensional		XML
Foundation	Business information	Data types	Expression	Keys and indexes	Type mapping	Software deployment
	UML 1.3 (Foundation, Common_Behavior, Model_Management)					

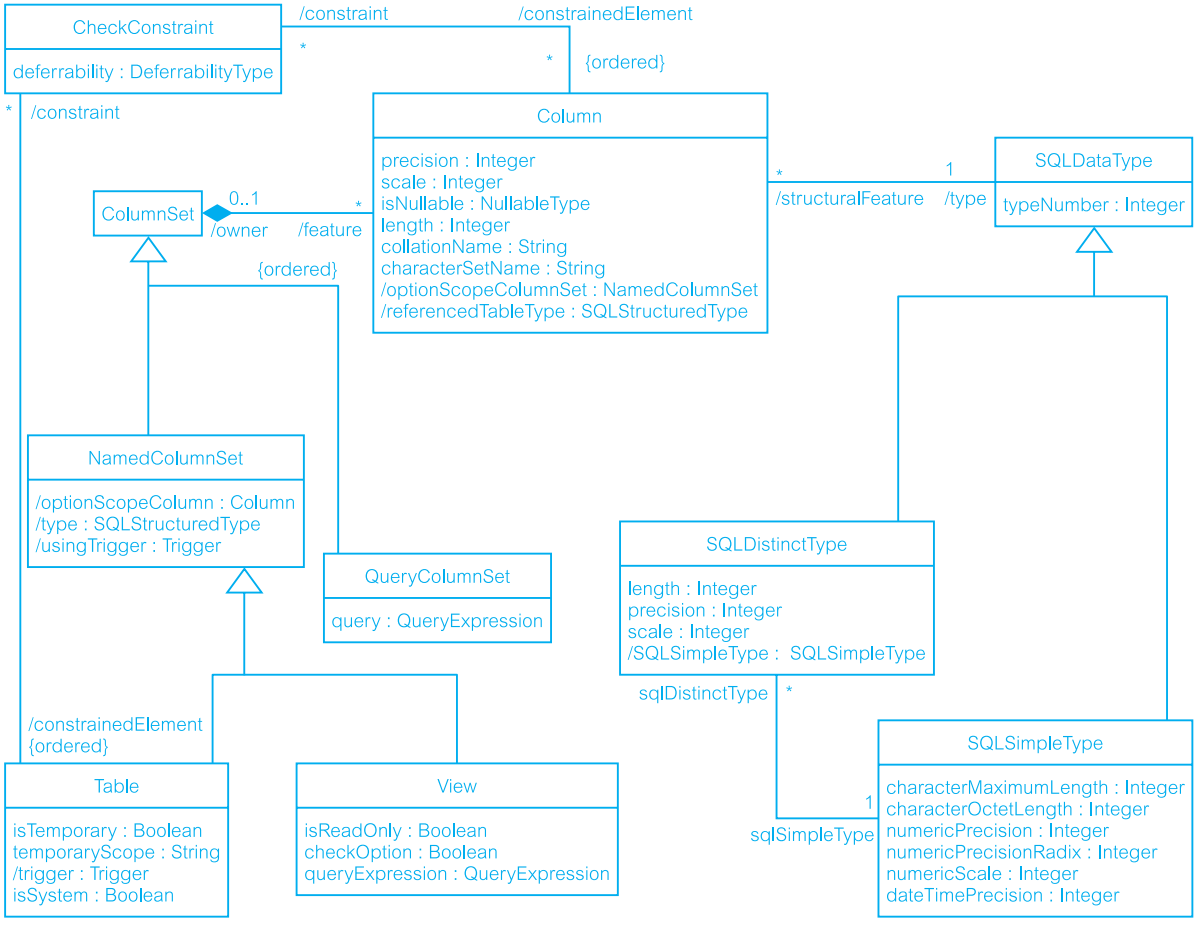


Figure 27.5 CWM Relational Data Metamodel.

Model-Driven Architecture 27.1.4

While the OMG hoped that the OMA would be embraced as the common object-oriented middleware standard, unfortunately other organizations developed alternatives. Microsoft produced the proprietary DCOM (Distributed Common Object Model), Sun developed Java, which came with its own ORB, Remote Method Invocation (RMI), and more recently another set of middleware standards emerged with XML and SOAP (Simple Object Office Access Protocol), which Microsoft, Sun, and IBM have all embraced. At the same time, the move towards e-Business increased the pressure on organizations to integrate their corporate databases. This integration, now termed **Enterprise Application Integration** (**EAI**), is one of the current key challenges for organizations and, rather than helping, it has been argued that middleware is part of the problem.

In 1999, the OMG started work on moving beyond OMA and CORBA and producing a new approach to the development of distributed systems. This work led to the introduction of the **Model-Driven Architecture** (**MDA**) as an approach to system specification and interoperability building upon the four modeling specifications discussed in the previous section. It is based on the premise that systems should be specified independent of all hardware and software details. Thus, whereas the software and hardware may change over time, the specification will still be applicable. Importantly, MDA addresses the complete system lifecycle from analysis and design to implementation, testing, component assembly, and deployment.

To create an MDA-based application, a Platform Independent Model (PIM) is produced that represents only business functionality and behavior. The PIM can then be mapped to one or more Platform Specific Models (PSMs) to target platforms like the CORBA Component Model (CCM), Enterprise JavaBeans (EJB), or Microsoft Transaction Server (MTS). Both the PIM and the PSM are expressed using the UML. The architecture encompasses the full range of pervasive services already specified by the OMG, such as Persistence, Transactions, and Security (see Table 27.1). Importantly, MDA enables the production of standardized domain models for specific vertical industries. The OMG will define a set of profiles to ensure that a given UML model can consistently generate each of the popular middleware APIs. Figure 27.6 illustrates how the various components in the MDA relate to each other.

Object Data Standard ODMG 3.0, 1999 27.2

In this section we review the new standard for the Object-Oriented Data Model (OODM) proposed by the Object Data Management Group (ODMG). It consists of an Object Model (Section 27.2.2), an Object Definition Language equivalent to the Data Definition Language (DDL) of a conventional DBMS (Section 27.2.3), and an Object Query Language with a SQL-like syntax (Section 27.2.4). We start with an introduction to the ODMG.

Object Data Management Group 27.2.1

Several important vendors formed the Object Data Management Group to define standards for OODBMSs. These vendors included Sun Microsystems, eXcelon Corporation,

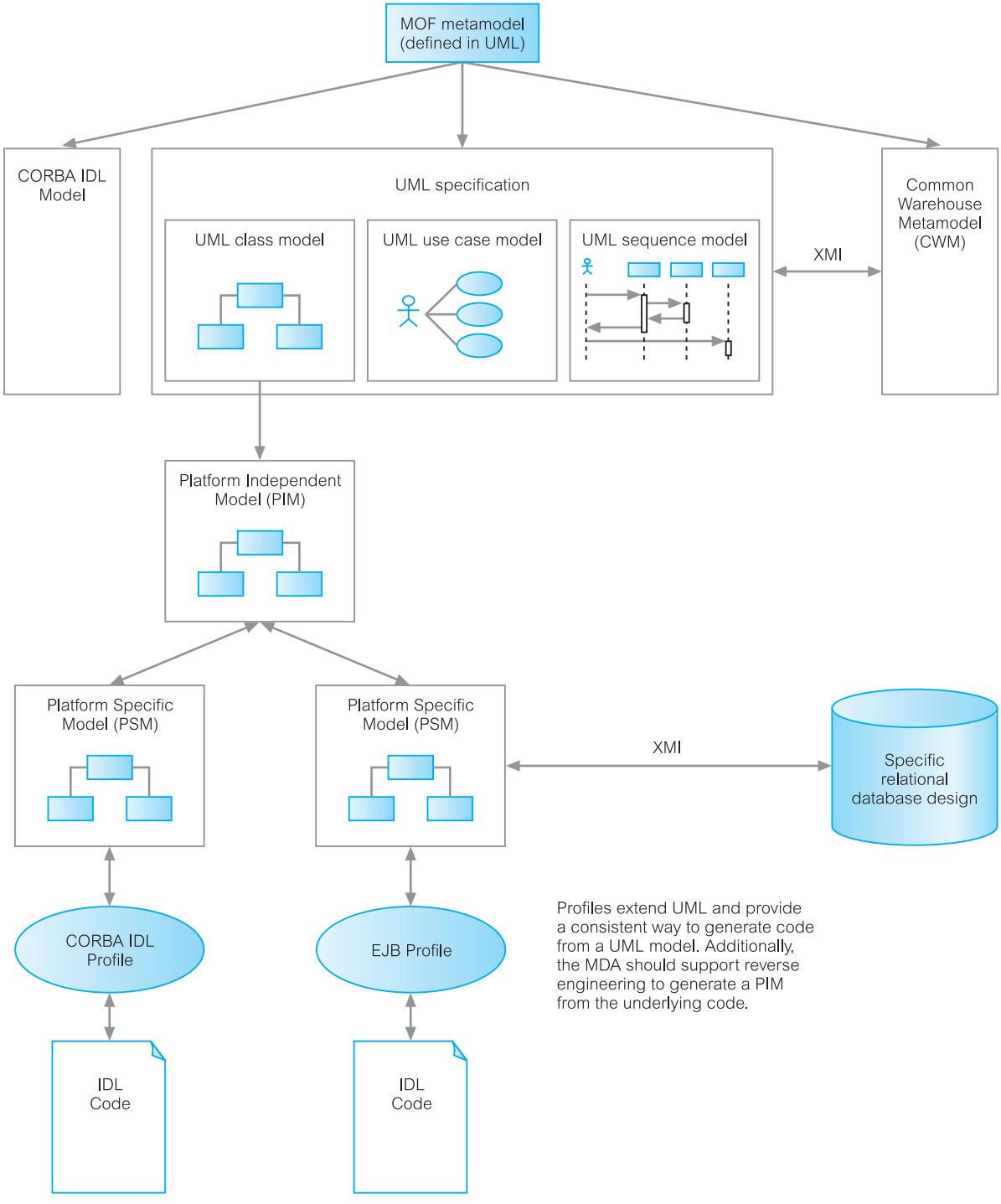


Figure 27.6 The Model-Driven Architecture.

Objectivity Inc., POET Software, Computer Associates, and Versant Corporation. The ODMG produced an object model that specifies a standard model for the semantics of database objects. The model is important because it determines the built-in semantics that the OODBMS understands and can enforce. As a result, the design of class libraries and applications that use these semantics should be portable across the various OODBMSs that support the object model (Connolly, 1994).

The major components of the ODMG architecture for an OODBMS are:

- Object Model (OM);
- Object Definition Language (ODL);
- Object Query Language (OQL);
- C++, Java, and Smalltalk language bindings.

We discuss these components in the remainder of this section. The initial version of the ODMG standard was released in 1993. There have been a number of minor releases since then, but a new major version, ODMG 2.0, was adopted in September 1997 with enhancements that included:

- a new binding for Sun's Java programming language;
- a fully revised version of the Object Model, with a new metamodel supporting object database semantics across many programming languages;
- a standard external form for data and the data schema allowing data interchanges between databases.

In late 1999, ODMG 3.0 was released that included a number of enhancements to the Object Model and to the Java binding. Between releases 2.0 and 3.0, the ODMG expanded its charter to cover the specification of *universal object storage standards*. At the same time, ODMG changed its name from the Object Database Management Group to the Object Data Management Group to reflect the expansion of its efforts beyond merely setting storage standards for object databases.

The ODMG Java binding was submitted to the Java Community Process as the basis for the Java Data Objects (JDO) Specification, although JDO is now based on a native Java language approach rather than a binding. A public release of the JDO specification is now available, which we discuss in Chapter 29. The ODMG completed its work in 2001 and disbanded.

Terminology

Under its last charter, the ODMG specification covers both OODBMSs that store objects directly and Object-to-Database Mappings (ODMs) that convert and store the objects in a relational or other database system representation. Both types of product are referred to generically as Object Data Management Systems (ODMSs). ODMSs make database objects appear as programming language objects in one or more existing (object-oriented) programming languages, and ODMSs extend the programming language with transparently persistent data, concurrency control, recovery, associative queries, and other database capabilities (Cattell, 2000).

27.2.2 The Object Model

The ODMG OM is a superset of the OMG OM, which enables both designs and implementations to be ported between compliant systems. It specifies the following basic modeling primitives:

- The basic modeling primitives are the **object** and the **literal**. Only an object has a unique identifier.
- Objects and literals can be categorized into **types**. All objects and literals of a given type exhibit common behavior and state. A type is itself an object. An object is sometimes referred to as an **instance** of its type.
- Behavior is defined by a set of **operations** that can be performed on or by the object. Operations may have a list of typed input/output parameters and may return a typed result.
- State is defined by the values an object carries for a set of **properties**. A property may be either an **attribute** of the object or a **relationship** between the object and one or more other objects. Typically, the values of an object's properties can change over time.
- An **ODMS** stores objects, enabling them to be shared by multiple users and applications. An ODMS is based on a **schema** that is defined in the **Object Definition Language** (ODL), and contains instances of the types defined by its schema.

Objects

An object is described by four characteristics: structure, identifier, name, and lifetime, as we now discuss.

Object structure

Object types are decomposed as atomic, collections, or structured types, as illustrated in Figure 27.7. In this structure, types shown in *italics* are abstract types; the types shown in normal typeface are directly instantiable. We can use only types that are directly instantiable as base types. Types with angle brackets < > indicate type generators. All atomic objects are user-defined whereas there are a number of built-in collection types, as we see shortly. As can be seen from Figure 27.7, the structured types are as defined in the ISO SQL specification (see Section 6.1).

Objects are created using the new method of the corresponding *factory interface* provided by the language binding implementation. Figure 27.8 shows the ObjectFactory interface, which has a new method to create a new instance of type Object. In addition, all objects have the ODL interface shown in Figure 27.8, which is implicitly inherited by the definitions of all user-defined object types.

Object identifiers and object names

Each object is given a unique identity by the ODMS, the *object identifier*, which does not change and is not reused when the object is deleted. In addition, an object may also be given one or more *names* that are meaningful to the user, provided each name identifies a single object within a database. Object names are intended to act as 'root' objects that

```
Literal_type
    Atomic_literal
        long
        long long
        short
        unsigned long
        unsigned short
        float
        double
        boolean
        octet
        char
        string
        enum<>              // enumeration
    Collection_literal
        set<>
        bag<>
        list<>
        array<>
        dictionary<>
    Structured_literal
        date
        time
        timestamp
        interval
        structure<>
Object_type
    Atomic_object
    Collection_object
        Set<>
        Bag<>
        List<>
        Array<>
        Dictionary<>
    Structured_object
        Date
        Time
        Timestamp
        Interval
```

Figure 27.7
Full set of built-in types for ODMG Object Model.

provide entry points into the database. As such, methods for naming objects are provided within the Database class (which we discuss shortly) and not within the object class.

Object lifetimes

The standard specifies that the lifetime of an object is orthogonal to its type, that is, persistence is independent of type (see Section 26.3.2). The lifetime is specified when the object is created and may be:

Figure 27.8
ODL interface for user-defined object types.

```
interface ObjectFactory {
    Object    new();
}
interface Object {
    enum      Lock_Type{read, write, upgrade};
    void      lock(in Lock_Type mode) raises(LockNotGranted); // obtain lock – wait if necessary
    boolean   try_lock(in Lock_Type mode);          // obtain lock – do not wait if not immediately granted
    boolean   same_as(in Object anObject);          // identity comparison
    Object    copy();                               // copy object – copied object not 'same as'
    void      delete();                             // delete object from database
};
```

- **transient** – the object's memory is allocated and deallocated by the programming language's runtime system. Typically, allocation will be stack-based for objects declared in the heading of a procedure, and static storage or heap-based for dynamic (process-scoped) objects;
- **persistent** – the object's storage is managed by the ODMS.

Literals

A literal is basically a constant value, possibly with a complex structure. Being a constant, the values of its properties may not change. As such, literals do not have their own identifiers and cannot stand alone as objects: they are embedded in objects and cannot be individually referenced. Literal types are decomposed as atomic, collections, structured, or null. Structured literals contain a fixed number of named heterogeneous elements. Each element is a <name, value> pair, where *value* may be any literal type. For example, we could define a structure Address as follows:

```
struct Address {
        string      street;
        string      city;
        string      postcode;
};
attribute Address branchAddress;
```

In this respect, a structure is similar to the **struct** or **record** type in programming languages. Since structures are literals, they may occur as the value of an attribute in an object definition. We shall see an example of this shortly.

Built-in collections

In the ODMG Object Model, a collection contains an arbitrary number of unnamed homogeneous elements, each of which can be an instance of an atomic type, another collection, or a literal type. The only difference between collection objects and collection literals is that collection objects have identity. For example, we could define the set of all branch

```
interface Iterator {
    exception   NoMoreElements{};
    exception   InvalidCollectionType{};
    boolean     is_stable();
    boolean     at_end();
    void        reset();
    Object      get_element() raises(NoMoreElements);
    void        next_position() raises(NoMoreElements);
    void        replace_element(in Object element) raises(InvalidCollectionType);
};
interface BidirectionalIterator : Iterator {
    boolean     at_beginning();
    void        previous_position() raises(NoMoreElements);
};
```

Figure 27.9
ODL interface for iterators.

```
interface Collection: Object {
    exception               InvalidCollection{};
    exception               ElementNotFound{Object element;};
    unsigned long           cardinality();                          // return number of elements
    boolean                 is_empty();                             // check if collection empty
    boolean                 is_ordered();                           // check if collection is ordered
    boolean                 allows_duplicates();                    // check if duplicates are allowed
    boolean                 contains_element(in Object element);    // check for specified element
    void                    insert_element(in Object element);      // insert specified element
    void                    remove_element(in Object element)
                            raises(ElementNotFound);                // remove specified element
    Iterator                create_iterator(in boolean stable);     // create forward only traversal iterator
    BidirectionalIterator   create_bidirectional_iterator(in boolean stable)
                            raises(InvalidCollectionType);          // create bi-directional iterator
    Object                  select_element(in string OQL-predicate);
    Iterator                select(in string OQL-predicate);
    boolean                 query(in string OQL-predicate, inout Collection result);
    boolean                 exists_element(in string OQL-predicate);
};
```

Figure 27.10
ODL interface for collections.

offices as a collection. Iteration over a collection is achieved by using an *iterator* that maintains the current position within the given collection. There are ordered and unordered collections. Ordered collections must be traversed first to last, or *vice versa*; unordered collections have no fixed order of iteration. Iterators and collections have the operations shown in Figures 27.9 and 27.10, respectively.

The stability of an iterator determines whether iteration is safe from changes made to the collection during the iteration. An iterator object has methods to position the iterator pointer at the first record, get the current element, and increment the iterator to the next element, among others. The model specifies five built-in collection subtypes:

Figure 27.11
ODL interface for the Set and Dictionary collections.

```
interface SetFactory : ObjectFactory {
  Set       new_of_size(in long size);                  // create a new Set object
};
class Set : Collection {
  attribute set<t> value;
  Set       create_union(in Set other_set);             // union of two sets
  Set       create_intersection(in Set other_set);      // intersection of two sets
  Set       create_difference(in Set other_set);        // set difference of two sets
  boolean   is_subset_of(in Set other_set);             // check if one set is subset of another
  boolean   is_proper_subset_of(in Set other_set);      // check if one set is proper subset of another
  boolean   is_superset_of(in Set other_set);           // check if one set is superset of another
  boolean   is_proper_superset_of(in Set other_set);    // check if one set is proper superset of another
};
interface DictionaryFactory : ObjectFactory {
  Dictionarynew_of_size(in long size);
};
class Dictionary : Collection {
  exception  DuplicateName{string key;};
  exception  KeyNotFound{Object key;};
  attribute  dictionary<t, v> value;
  void       bind(in Object key, in Object value) raises(DuplicateName);
  void       unbind(in Object key) raises(KeyNotFound);
  void       lookup(in Object key) raises(KeyNotFound);
  void       contains_key(in Object key);
};
```

- Set – unordered collections that do not allow duplicates;
- Bag – unordered collections that do allow duplicates;
- List – ordered collections that allow duplicates;
- Array – one-dimensional array of dynamically varying length;
- Dictionary – unordered sequence of key-value pairs with no duplicate keys.

Each subtype has operations to create an instance of the type and insert an element into the collection. Sets and Bags have the usual set operations: union, intersection, and difference. The interface definitions for the Set and Dictionary collections are shown in Figure 27.11.

Atomic objects

Any user-defined object that is not a collection object is called an **atomic object**. For example, for *DreamHome* we will want to create atomic object types to represent Branch and Staff. Atomic objects are represented as a **class**, which comprises *state* and *behavior*. **State** is defined by the values an object carries for a set of *properties*, which may be either an *attribute* of the object or a *relationship* between the object and one or more other objects.

Behavior is defined by a set of *operations* that can be performed on or by the object. In addition, atomic objects can be related in a supertype/subtype lattice. As expected, a subtype inherits all the attributes, relationships, and operations defined on the supertype, and may define additional properties and operations and redefine inherited properties and operations. We now discuss attributes, relationships, and operations in more detail.

Attributes

An attribute is defined on a single object type. An attribute is not a 'first class' object, in other words it is not an object and so does not have an object identifier, but takes as its value a literal or an object identifier. For example, a Branch class has attributes for the branch number, street, city, and postcode.

Relationships

Relationships are defined between types. However, the model supports only binary relationships with cardinality 1:1, 1:*, and *:*. A relationship does not have a name and, again, is not a 'first class' object; instead, **traversal paths** are defined for each direction of traversal. For example, a Branch *Has* a set of Staff and a member of Staff *WorksAt* a Branch, would be represented as:

```
class Branch {
        relationship set <Staff> Has inverse Staff::WorksAt;
};
class Staff {
        relationship Branch WorksAt inverse Branch::Has;
};
```

On the many side of relationships, the objects can be unordered (a Set or Bag) or ordered (a List). Referential integrity of relationships is maintained automatically by the ODMS and an exception (that is, an error) is generated if an attempt is made to traverse a relationship in which one of the participating objects has been deleted. The model specifies built-in operations to form and drop members from relationships, and to manage the required referential integrity constraints. For example, the 1:1 relationship Staff *WorksAt* Branch would result in the following definitions on the class Staff for the relationship with Branch:

```
attribute Branch WorksAt;
void      form_WorksAt(in Branch aBranch) raises(IntegrityError);
void      drop_WorksAt(in Branch aBranch) raises(IntegrityError);
```

The 1:* relationship Branch *Has* Staff would result in the following definitions on the class Branch for the relationship with Staff:

```
readonly  attribute  set <Staff> Has;
void      form_Has(in Staff aStaff) raises(IntegrityError);
void      drop_Has(in Staff aStaff) raises(IntegrityError);
void      add_Has(in Staff aStaff) raises(IntegrityError);
void      remove_Has(in Staff aStaff) raises(IntegrityError);
```

Operations

The instances of an object type have behavior that is specified as a set of operations. The object type definition includes an **operation signature** for each operation that specifies the name of the operation, the names and types of each argument, the names of any exceptions that can be raised, and the types of the values returned, if any. An operation can be defined only in the context of a single object type. Overloading operation names is supported. The model assumes sequential execution of operations and does not require support for concurrent, parallel, or remote operations, although it does not preclude such support.

Types, classes, interfaces, and inheritance

In the ODMG Object Model there are two ways to specify object types: interfaces and classes. There are also two types of inheritance mechanism, as we now discuss.

An **interface** is a specification that defines only the abstract behavior of an object type, using operation signatures. *Behavior inheritance* allows interfaces to be inherited by other interfaces and classes using the ':' symbol. Although an interface may include properties (attributes and relationships), these cannot be inherited from the interface. An interface is also non-instantiable, in other words we cannot create objects from an interface (in much the same way as we cannot create objects from a C++ abstract class). Normally, interfaces are used to specify abstract operations that can be inherited by classes or by other interfaces.

On the other hand, a **class** defines both the abstract state and behavior of an object type, and is instantiable (thus, interface is an abstract concept and class is an implementation concept). We can also use the **extends** keyword to specify single inheritance between classes. Multiple inheritance is not allowed using *extends* although it is allowed using behavior inheritance. We shall see examples of both these types of inheritance shortly.

Extents and keys

A class definition can specify its **extent** and its **keys**:

- **Extent** is the set of all instances of a given type within a particular ODMS. The programmer may request that the ODMS maintain an index to the members of this set. Deleting an object removes the object from the extent of a type of which it is an instance.
- **Key** uniquely identifies the instances of a type (similar to the concept of a candidate key defined in Section 3.2.5). A type must have an extent to have a key. Note also, that a key is different from an object name: a key is composed of properties specified in an object type's interface whereas an object name is defined within the database type.

Exceptions

The ODMG model supports dynamically nested exception handlers. As we have already noted, operations can raise exceptions and exceptions can communicate exception results. Exceptions are 'first class' objects that can form a generalization–specialization hierarchy, with the root type Exception provided by the ODMS.

Metadata

As we discussed in Section 2.4, metadata is 'the data about data': that is, data that describes objects in the system, such as classes, attributes, and operations. Many existing ODMSs do not treat metadata as objects in their own right, and so a user cannot query the metadata as they can query other objects. The ODMG model defines metadata for:

- *scopes*, which define a naming hierarchy for the meta-objects in the repository;
- *meta-objects*, which consist of modules, operations, exceptions, constants, properties (consisting of attributes and relationships), and types (consisting of interfaces, classes, collections, and constructed types);
- *specifiers*, which are used to assign a name to a type in certain contexts;
- *operands*, which form the base type for all constant values in the repository.

Transactions

The ODMG Object Model supports the concept of transactions as logical units of work that take the database from one consistent state to another (see Section 20.1). The model assumes a linear sequence of transactions executing within a thread of control. Concurrency is based on standard read/write locks in a pessimistic concurrency control protocol. All access, creation, modification, and deletion of persistent objects must be performed within a transaction. The model specifies built-in operations to begin, commit, and abort transactions, as well as a checkpoint operation, as shown in Figure 27.12. A checkpoint commits all modified objects in the database without releasing any locks before continuing the transaction.

The model does not preclude distributed transaction support but states that if it is provided it must be XA-compliant (see Section 23.5).

Databases

The ODMG Object Model supports the concept of databases as storage areas for persistent objects of a given set of types. A database has a schema that contains a set of type

```
interface TransactionFactory {
    Transaction   new();
    Transaction   current();
};
interface Transaction {
    void        begin() raises(TransactionInProgress, DatabaseClosed);
    void        commit() raises(TransactionNotInProgress);
    void        abort() raises(TransactionNotInProgress);
    void        checkpoint() raises(TransactionNotInProgress);
    void        join() raises(TransactionNotInProgress);
    void        leave() raises(TransactionNotInProgress);
    boolean     isOpen();
};
```

Figure 27.12
ODL interface for transactions.

Figure 27.13
ODL interface for database objects.

```
interface DatabaseFactory {
    Database    new();
};
interface Database{
    exception   DatabaseOpen{};
    exception   DatabaseNotFound{};
    exception   ObjectNameNotUnique{};
    exception   ObjectNameNotFound{};
    void        open(in string odms_name) raises(DatabaseNotFound, DatabaseOpen);
    void        close() raises(DatabaseClosed, TransactionInProgress);
    void        bind(in Object an_object, in string name) raises(DatabaseClosed,
                                    ObjectNameNotUnique, TransactionNotInProgress);
    void        unbind(in string name) raises(DatabaseClosed,
                                    ObjectNameNotFound, TransactionNotInProgress);
    void        lookup(in string object_name) raises(DatabaseClosed,
                                    ObjectNameNotFound, TransactionNotInProgress);
    ODLMetaObjects::Module schema() raises(DatabaseClosed, TransactionNotInProgress);
};
```

definitions. Each database is an instance of type Database with the built-in operations open and close, and lookup, which checks whether a database contains a specified object. Named objects are entry points to the database, with the name bound to an object using the built-in bind operation, and unbound using the unbind operation, as shown in Figure 27.13.

Modules

Parts of a schema can be packaged together to form named modules. Modules have two main uses:

- they can be used to group together related information so that it can be handled as a single, named entity;
- they can be used to establish the scope of declarations, which can be useful to resolve naming conflicts that may arise.

27.2.3 The Object Definition Language

The Object Definition Language (ODL) is a language for defining the specifications of object types for ODMG-compliant systems, equivalent to the Data Definition Language (DDL) of traditional DBMSs. Its main objective is to facilitate portability of schemas between compliant systems while helping to provide interoperability between ODMSs. The ODL defines the attributes and relationships of types and specifies the signature of the operations, but it does not address the implementation of signatures. The syntax of ODL extends the Interface Definition Language (IDL) of CORBA. The ODMG hoped that the

ODL would be the basis for integrating schemas from multiple sources and applications. A complete specification of the syntax of ODL is beyond the scope of this book. However, Example 27.1 illustrates some of the elements of the language. The interested reader is referred to Cattell (2000) for a complete definition.

Example 27.1 The Object Definition Language

Consider the simplified property for rent schema for *DreamHome* shown in Figure 27.14. An example ODL definition for part of this schema is shown in Figure 27.15.

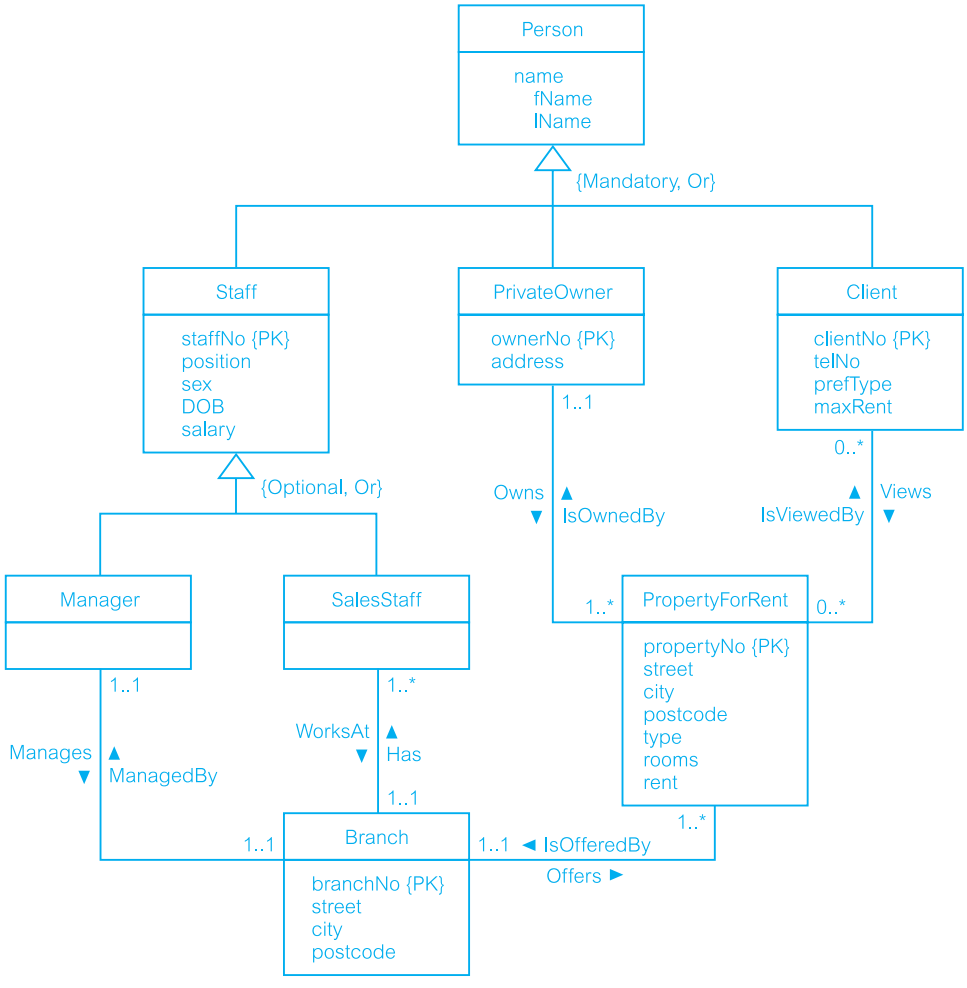


Figure 27.14 Example *DreamHome* property for rent schema.

Figure 27.15
ODL definition for part of the *DreamHome* property for rent schema.

```
module DreamHome {
    class Branch                                    // Define class for Branch
     (extent branchOffices   key branchNo)
    {
    /* Define attributes */
     attribute string branchNo;
     attribute struct BranchAddress {string street, string city, string postcode} address;
    /* Define relationships */
     relationship Manager ManagedBy inverse Manager::Manages;
     relationship set<SalesStaff> Has inverse SalesStaff::WorksAt;
     relationship set<PropertyForRent> Offers inverse PropertyForRent::IsOfferedBy;
    /* Define operations */
     void takeOnPropertyForRent(in string propertyNo) raises(propertyAlreadyForRent);
    };

    class Person {                                  // Define class for Person
    /* Define attributes */
     attribute struct PName {string fName, string lName} name;
    };

    class Staff extends Person                      // Define class for Staff that inherits from Person
     (extent staff   key staffNo)
    {
    /* Define attributes */
     attribute string staffNo;
     attribute enum SexType {M, F} sex;
     attribute enum PositionType {Manager, Supervisor, Assistant} position;
     attribute date DOB;
     attribute float salary;
    /* Define operations */
     short getAge();
     void increaseSalary(in float increment);
    };

    class Manager extends Staff                     // Define class for Manager that inherits from Staff
     (extent managers)
    {
    /* Define relationships */
     relationship Branch Manages inverse Branch::ManagedBy;
    };

    class SalesStaff extends Staff                  // Define class for SalesStaff that inherits from Staff
     (extent salesStaff)
    {
    /* Define relationships */
     relationship Branch WorksAt inverse Branch::Has;
    /* Define operations */
     void transferStaff(in string fromBranchNo, in string toBranchNo) raises(doesNotWorkInBranch);
    };
};
```

The Object Query Language 27.2.4

The Object Query Language (OQL) provides declarative access to the object database using an SQL-like syntax. It does not provide explicit update operators, but leaves this to the operations defined on object types. As with SQL, OQL can be used as a standalone language and as a language embedded in another language for which an ODMG binding is defined. The supported languages are Smalltalk, C++, and Java. OQL can also invoke operations programmed in these languages.

OQL can be used both for both associative and navigational access:

- An associative query returns a collection of objects. How these objects are located is the responsibility of the ODMS, rather than the application program.
- A navigational query accesses individual objects and object relationships are used to navigate from one object to another. It is the responsibility of the application program to specify the procedure for accessing the required objects.

An OQL query is a function that delivers an object whose type may be inferred from the operator contributing to the query expression. Before we expand on this definition, we first have to understand the composition of expressions. For this section it is assumed that the reader is familiar with the functionality of the SQL SELECT statement covered in Section 5.3.

Expressions

Query definition expression

A query definition expression is of the form: **DEFINE** Q **AS** e. This defines a **named query** (that is, *view*) with name *Q*, given a query expression *e*.

Elementary expressions

An expression can be:

- an atomic literal, for example, 10, 16.2, 'x', 'abcde', true, nil, date'2004-12-01';
- a named object, for example, the extent of the Branch class, branchOffices in Figure 27.15, is an expression that returns the set of all branch offices;
- an iterator variable from the FROM clause of a SELECT-FROM-WHERE statement, for example,

  e **AS** x or e x or x **IN** e

  where e is of type collection (T), then x is of type T. We discuss the OQL SELECT statement shortly;
- a query definition expression (*Q* above).

Construction expressions

- If T is a type name with properties $p_1, \ldots, p_n$, and $e_1, \ldots, e_n$ are expressions, then $T(p_1{:}e_1, \ldots p_n{:}e_n)$ is an expression of type T. For example, to create a Manager object, we could use the following expression:

Manager(staffNo: "SL21", fName: "John", lName: "White",
address: "19 Taylor St, London", position: "Manager", sex: "M",
DOB: date'1945-10-01', salary: 30000)

- Similarly, we can construct expressions using struct, Set, List, Bag, and Array. For example:

  struct(branchNo: "B003", street: "163 Main St")

  is an expression, which dynamically creates an instance of this type.

Atomic type expressions

Expressions can be formed using the standard unary and binary operations on expressions. Further, if S is a string, expressions can be formed using:

- standard unary and binary operators, such as **not**, **abs**, +, –, =, >, **andthen**, **and**, **orelse**, **or**;
- the string concatenation operation (|| or +);
- a string offset Si (where i is an integer) meaning the *i+1*th character of the string;
- S[low:up] meaning the substring of S from the *low+1*th to *up+1*th character;
- 'c in S' (where c is a character), returning a boolean true expression if the character c is in S;
- 'S like *pattern*', where *pattern* contains the characters '?' or '_', meaning any character, or the wildcard characters '*' or '%', meaning any substring including the empty string. This returns a boolean true expression if S matches the pattern.

Object expressions

Expressions can be formed using the equality and inequality operations ('=' and '!='), returning a boolean value. If e is an expression of a type having an attribute or a relationship p of type T, then we can extract the attribute or traverse the relationship using the expressions e.p and e → p, which are of type T.

In a same way, methods can be invoked to return an expression. If the method has no parameters, the brackets in the method call can be omitted. For example, the method getAge() of the class Staff can be invoked as getAge (without the brackets).

Collections expressions

Expressions can be formed using universal quantification (**FOR ALL**), existential quantification (**EXISTS**), membership testing (**IN**), select clause (**SELECT FROM WHERE**), order-by operator (**ORDER BY**), unary set operators (**MIN, MAX, COUNT, SUM, AVG**), and the group-by operator (**GROUP BY**). For example,

FOR ALL x **IN** managers: x.salary > 12000

returns true for all the objects in the extent managers with a salary greater than £12,000. The expression:

EXISTS x **IN** managers.manages: x.address.city = "London";

returns true if there is at least one branch in London (managers.manages returns a Branch object and we then check whether the city attribute of this object contains the value London).

The format of the SELECT clause is similar to the standard SQL SELECT statement (see Section 5.3.1):

```
SELECT [DISTINCT]   <expression>
FROM                <fromList>
[WHERE              <expression>]
[GROUP BY           <attribute1:expression1, attribute2:expression2,...>]
                    [HAVING <predicate>]
[ORDER BY           <expression>]
```

where:

```
<fromList> ::= <variableName> IN <expression> |
               <variableName> IN <expression>, <fromList> |
               <expression> AS <variableName> |
               <expression> AS <variableName>, <fromList>
```

The result of the query is a Set for SELECT DISTINCT, a List if ORDER BY is used, and a Bag otherwise. The ORDER BY, GROUP BY, and HAVING clauses have their usual SQL meaning (see Sections 5.3.2 and 5.3.4). However, in OQL the functionality of the GROUP BY clause has been extended to provide an explicit reference to the collection of objects within each group (which in OQL is called a *partition*), as we illustrate below in Example 27.6.

Conversion expressions

- If e is an expression, then element(e) is an expression that checks e is a singleton, raising an exception if it is not.
- If e is a list expression, then listtoset(e) is an expression that converts the list into a set.
- If e is a collection-valued expression, then flatten(e) is an expression that converts a collection of collections into a collection, that is, it flattens the structure.
- If e is an expression and c is a type name, then c(e) is an expression that asserts e is an object of type c, raising an exception if it is not.

Indexed collections expressions

If e_1, e_2 are lists or arrays and e_3, e_4 are integers, then $e_1[e_3]$, $e_1[e_3: e_4]$, first(e_1), last(e_1), and ($e_1 + e_2$) are expressions. For example:

```
first(element(SELECT b FROM b IN branchOffices
              WHERE b.branchNo = "B001").Has);
```

returns the first member of the set of sales staff at branch B001.

Binary set expressions

If e_1, e_2 are sets or bags, then the set operators union, except, and intersect of e_1 and e_2 are expressions.

Queries

A query consists of a (possibly empty) set of query definition expressions followed by an expression. The result of a query is an object with or without identity.

Example 27.2 Object Query Language – use of extents and traversal paths

(1) Get the set of all staff (with identity).

In general, an entry point to the database is required for each query, which can be any named persistent object (that is, an *extent* or a *named object*). In this case, we can use the extent of class Staff to produce the required set using the following simple expression:

```
staff
```

(2) Get the set of all branch managers (with identity).

```
branchOffices.ManagedBy
```

In this case, we can use the name of the extent of the class Branch (branchOffices) as an entry point to the database and then use the relationship ManagedBy to find the set of branch managers.

(3) Find all branches in London.

```
SELECT b.branchNo
FROM b IN branchOffices
WHERE b.address.city = "London";
```

Again, we can use the extent branchOffices as an entry point to the database and use the iterator variable b to range over the objects in this collection (similar to a tuple variable that ranges over tuples in the relational calculus). The result of this query is of type bag<string>, as the select list contains only the attribute branchNo, which is of type string.

(4) Assume that londonBranches is a named object (corresponding to the object from the previous query). Use this named object to find all staff who work at that branch.

We can express this query as:

```
londonBranches.Has
```

which returns a set<SalesStaff>. To access the salaries of sales staff, intuitively we may think this can be expressed as:

```
londonBranches.Has.salary
```

However, this is not allowed in OQL because there is ambiguity over the return result: it may be set<float> or bag<float> (bag would be more likely because more than one member of staff may have the same salary). Instead, we have to express this as:

```
SELECT [DISTINCT] s.salary
FROM s IN londonBranches.Has;
```

Specifying DISTINCT would return a set<float> and omitting DISTINCT would return bag<float>.

Example 27.3 Object Query Language – use of DEFINE

Get the set of all staff who work in London (without identity).

We can express this query as:

```
DEFINE Londoners AS
        SELECT s
        FROM s IN salesStaff
        WHERE s.WorksAt.address.city = "London";
SELECT s.name.lName FROM s IN Londoners;
```

which returns a literal of type set<string>. In this example, we have used the DEFINE statement to create a view in OQL and then queried this view to obtain the required result. In OQL, the name of the view must be a unique name among all named objects, classes, methods, or function names in the schema. If the name specified in the DEFINE statement is the same as an existing schema object, the new definition replaces the previous one. OQL also allows a view to have parameters, so we can generalize the above view as:

```
DEFINE CityWorker(cityname) AS
        SELECT s
        FROM s IN salesStaff
        WHERE s.WorksAt.address.city = cityname;
```

We can now use the above query to find staff in London and Glasgow as follows:

```
CityWorker("London");
CityWorker("Glasgow");
```

Example 27.4 Object Query Language – use of structures

(1) Get the structured set (without identity) containing the name, sex, and age of all sales staff who work in London.

We can express this query as:

```
SELECT struct (lName: s.name.lName, sex: s.sex, age: s.getAge)
FROM s IN salesStaff
WHERE s.WorksAt.address.city = "London";
```

which returns a literal of type set<struct>. Note in this case the use of the method getAge in the SELECT clause.

(2) Get the structured set (with identity) containing the name, sex, and age of all deputy managers over 60.

We can express this query as:

```
class Deputy {attribute string lName; attribute sexType sex; attribute integer age;};
typedef bag<Deputy> Deputies;
Deputies (SELECT Deputy (lName: s.name.lname, sex: s.sex, age: s.getAge)
          FROM s IN staffStaff WHERE position = "Deputy" AND s.getAge > 60);
```

which returns a mutable object of type Deputies.

(3) Get a structured set (without identity) containing the branch number and the set of all Assistants at the branches in London.

The query, which returns a literal of type set<struct>, is:

```
SELECT struct (branchNo: x.branchNo, assistants: (SELECT y FROM y
                IN x.WorksAt WHERE y.position = "Assistant"))
FROM x IN (SELECT b FROM b IN branchOffices
           WHERE b.address.city = "London");
```

Example 27.5 Object Query Language – use of aggregates

How many staff work in Glasgow?

In this case, we can use the aggregate operation COUNT and the view CityWorker defined earlier to express this query as:

```
COUNT(s IN CityWorker("Glasgow"));
```

The OQL aggregate functions can be applied within the select clause or to the result of the select operation. For example, the following two expressions are equivalent in OQL:

```
SELECT COUNT(s) FROM s IN salesStaff WHERE s.WorksAt.branchNo = "B003";
COUNT(SELECT s FROM s IN salesStaff WHERE s.WorksAt.branchNo = "B003");
```

Note that OQL allows aggregate operations to be applied to any collection of the appropriate type and, unlike SQL, can be used in any part of the query. For example, the following is allowed in OQL (but not SQL):

```
SELECT s
FROM s IN salesStaff
WHERE COUNT(s.WorksAt) > 10;
```

Example 27.6 GROUP BY and HAVING clauses

Determine the number of sales staff at each branch.

```
SELECT struct(branchNumber, numberOfStaff: COUNT(partition))
FROM s IN salesStaff
GROUP BY branchNumber: s.WorksAt.branchNo;
```

The result of the grouping specification is of type set<struct(branchNumber: string, partition: bag<struct(s: SalesStaff)>)>, which contains a struct for each partition (group) with two components: the grouping attribute value branchNumber and a bag of the sales staff objects in the partition. The SELECT clause then returns the grouping attribute, branchNumber, and a count of the number of elements in each partition (in this case, the number of sales staff in each branch). Note the use of the keyword **partition** to refer to each partition. The overall result of this query is:

```
set<struct(branchNumber: string, numberOfStaff: integer)>
```

As with SQL, the HAVING clause can be used to filter the partitions. For example, to determine the average salary of sales staff for those branches with more than ten sales staff we could write:

```
SELECT branchNumber, averageSalary: AVG(SELECT p.s.salary FROM p IN partition)
FROM s IN salesStaff
GROUP BY branchNumber: s.WorksAt.branchNo
HAVING COUNT(partition) > 10;
```

Note the use of the SELECT statement within the aggregate operation AVG. In this statement, the iterator variable p iterates over the partition collection (of type bag<struct(s: SalesStaff)>). The path expression p.s.salary is used to access the salary of each sales staff member in the partition.

Other Parts of the ODMG Standard 27.2.5

In this section, we briefly discuss two other parts of the ODMG 3.0 standard:

- the Object Interchange Format;
- the ODMG language bindings.

Object Interchange Format

The Object Interchange Format (OIF) is a specification language used to dump and load the current state of an ODMS to and from one or more files. OIF can be used to exchange persistent objects between ODMSs, seed data, provide documentation, and drive test suites (Cattell, 2000). OIF was designed to support all possible ODMS states compliant with the ODMG Object Model and ODL schema definitions. It was also designed according to

NCITS (National Committee for Information Technology Standards) and PDES/STEP (Product Data Exchange using STEP, the STandard for the Exchange of Product model data) for mechanical CAD, wherever possible.

An OIF file is made up of one or more object definitions, where an object definition is an object identifier (with optional physical clustering indicator) and a class name (with optional initialization information). Some examples of object definitions are:

John {SalesStaff}	an instance of class SalesStaff is created with name John.
John (Mary) {SalesStaff}	an instance of class SalesStaff is created with name John physically near to the persistent object Mary. In this context, 'physically near' is implementation-dependent.
John SalesStaff{WorksAt B001}	creates a relationship called WorksAt between the instance John of class SalesStaff and the object named B001.

A complete description of the OIF specification language is beyond the scope of this book, but the interested reader is referred to Cattell (2000).

ODMG language bindings

The language bindings specify how ODL/OML constructs are mapped to programming language constructs. The languages supported by ODMG are C++, Java, and Smalltalk. The basic design principle for the language bindings is that the programmer should think there is only one language being used, not two separate languages. In this section we briefly discuss how the C++ binding works.

A C++ class library is provided containing classes and functions that implement the ODL constructs. In addition, OML (Object Manipulation Language) is used to specify how database objects are retrieved and manipulated within the application program. To create a working application, the C++ ODL declarations are passed through a C++ ODL preprocessor, which has the effect of generating a C++ header file containing the object database definition and storing the ODMS metadata in the database. The user's C++ application, which contains OML, is then compiled in the normal way along with the generated object database definition C++ header file. Finally, the object code output by the compiler is linked with the ODMS runtime library to produce the required executable image, as illustrated in Figure 27.16. In addition to the ODL/OML bindings, within ODL and OML the programmer can use a set of constructs, called **physical pragmas**, to control some physical storage characteristics such as the clustering of objects on disk, indexes, and memory management.

In the C++ class library, features that implement the interface to the ODMG Object Model are prefixed d_. Examples are d_Float, d_String, d_Short for base data types and d_List, d_Set, and d_Bag for collection types. There is also a class d_Iterator for the Iterator class and a class d_Extent for class extents. In addition, a template class d_Ref(T) is defined for each class T in the database schema that can refer to both persistent and transient objects of class T.

Relationships are handled by including either a reference (for a 1:1 relationship) or a collection (for a 1:* relationship). For example, to represent the 1:* *Has* relationship in the Branch class, we would write:

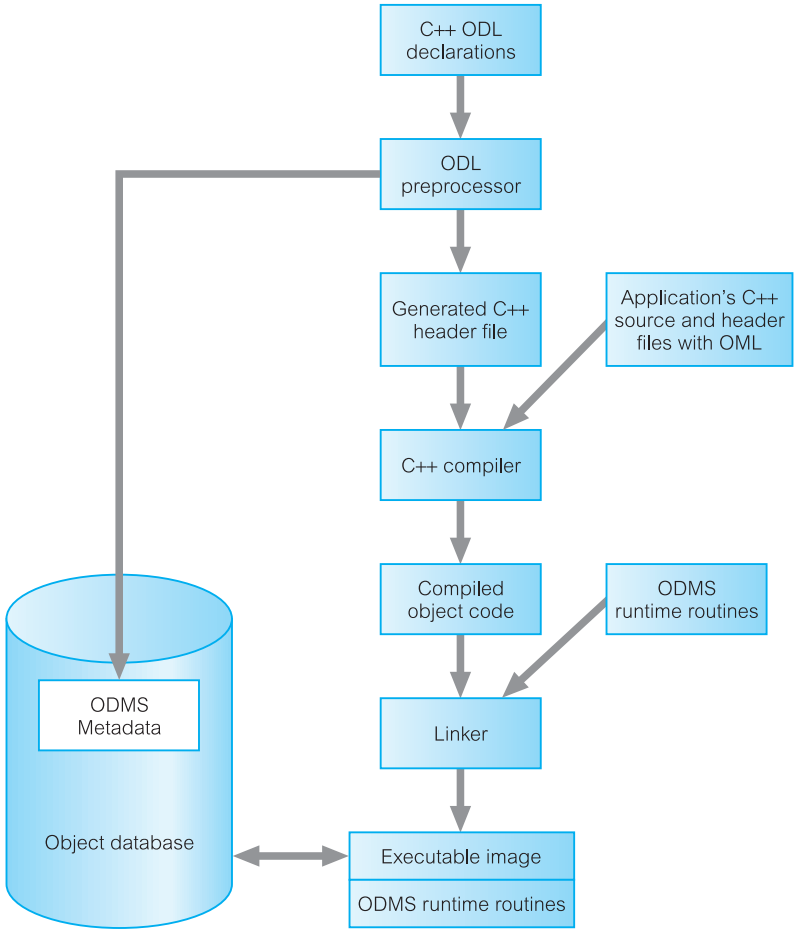


Figure 27.16
Compiling and linking a C++ ODL/OML application.

```
d_Rel_Set<SalesStaff, _WorksAt> Has;
const char _WorksAt[ ] = "WorksAt";
```

and to represent the same relationship in the SalesStaff class we would write:

```
d_Rel_Ref<Branch, _Has> WorksAt;
const char _Has[ ] = "Has";
```

Object Manipulation Language

For the OML, the new operator is overloaded so that it can create persistent or transient objects. To create a persistent object, a database name and a name for the object must be provided. For example, to create a transient object, we would write:

```
d_Ref<SalesStaff> tempSalesStaff = new SalesStaff;
```

and to create a persistent object we would write:

```
d_Database *myDB;
d_Ref<SalesStaff> s1 = new(myDb, "John White") SalesStaff;
```

Object Query Language

OQL queries can be executed from within C++ ODL/OML programs in one of the following ways:

- using the query member function of the d_Collection class;
- using the d_OQL_Query interface.

As an example of the first method, to obtain the set of sales staff (wellPaidStaff) with a salary greater than £30,000, we would write:

```
d_Bag<d_Ref<SalesStaff>> wellPaidStaff;
SalesStaff->query(wellPaidStaff, "salary > 30000");
```

As an example of the second method, to find the branches with sales staff who earn a salary above a specified threshold we would write:

```
d_OQL_Query q("SELECT s.WorksAt FROM s IN SalesStaff WHERE salary > $1");
```

This is an example of a parameterized query with $1 representing the runtime parameter. To specify a value for this parameter and run the query we would write:

```
d_Bag<d_Ref<Branch>> branches;
q << 30000;
d_oql_execute(q, branches);
```

For full details of the ODMG language bindings, the interested reader is referred to Cattell (2000).

27.2.6 Mapping the Conceptual Design to a Logical (Object-Oriented) Design

In Section 25.7.2 we briefly discussed how to use the various UML diagram types within a database design methodology. In this section we discuss how to map the conceptual schema to ODL. We assume that a class diagram has been produced as part of conceptual database design, consisting of classes (entity types), subclasses, attributes, methods, and a set of relationships.

Step 1 Mapping classes

Map each class or subclass to an ODL class, including all the appropriate attributes and methods. Map composite attributes to a tuple constructor using a *struct* declaration. Map any multivalued attributes as follows:

- if the values are ordered, map to a list constructor;
- if the values contain duplicates, map to a bag constructor;
- otherwise, map to a set constructor.

Create an extent for each class that will be iterated over. Specify EXTENDS for each ODL class that represents a subclass to inherit the attributes and methods of the superclass.

Step 2 Mapping binary relationships

For each binary relationship, add a relationship property (or reference attribute) into each class that participates in the relationship. If supported by the OODBMS, use inverse relationships where possible to ensure the system automatically maintains referential integrity. If the system does not support this, it will be necessary to program this functionality into the class methods.

If the multiplicity is 1:1, each relationship property will be single-valued; if it is 1:*, the relationship property will be single-valued on one side and a collection type (list or set depending upon the particular requirements of the relationship) on the other; if it is *:*, each side of the relationship will be a collection type (see Section 25.6).

Create a tuple constructor (struct) for relationship attributes of the form <relationship reference, relationship attributes>. This constructor is used in place of the relationship property. Unfortunately, this prevents inverse relationships from being used. Further, redundancy will exist if the relationship property is created in both directions.

Step 3 Mapping *n*-ary relationships

For each relationship with degree greater than 2 (for example, ternary, quaternary), create a separate class to represent the relationship and include a relationship property (based on a 1:* relationship) to each participating class.

Step 4 Mapping categories

For each category (union type) present in the class diagram create a class to represent the category and define a 1:1 relationship between the category class and each of its superclasses. Alternatively, a union type can be used if the OODBMS supports this.

27.3 ObjectStore

In this section we discuss the architecture and functionality of ObjectStore, a commercial OODBMS.

27.3.1 Architecture

ObjectStore is based on the multi-client/multi-server architecture, with each server responsible for controlling access to an object store and for managing concurrency control (locking-based), data recovery, and the transaction log, among other tasks. A client can contact the ObjectStore server on its host or any other ObjectStore server on any other host in the network. For each host machine running one or more client applications there is an associated **cache manager** process whose primary function is to facilitate concurrent access to data by handling callback messages from the server to client applications. In addition, each client application has its own **client cache**, which acts as a holding area for data mapped (or waiting to be mapped) into physical memory. A typical architecture

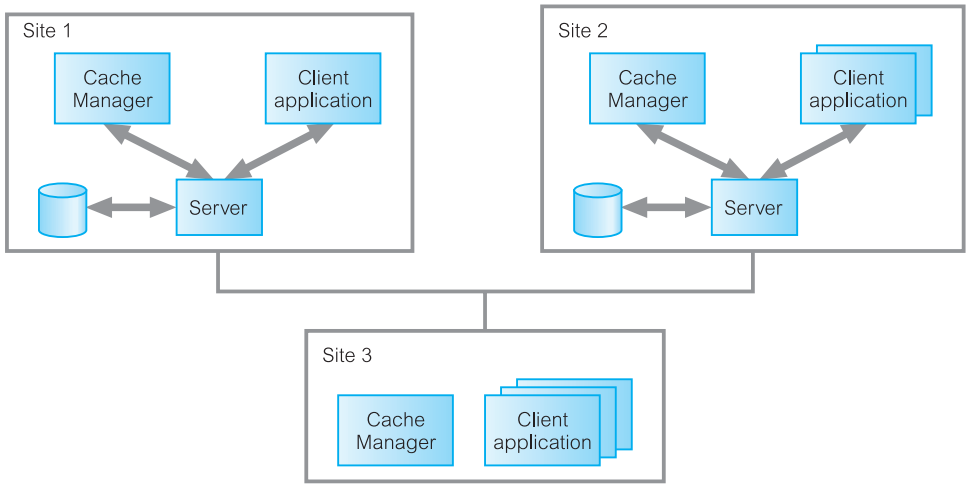


Figure 27.17 ObjectStore architecture.

is shown in Figure 27.17. We now briefly describe the main responsibilities of each of these processes.

ObjectStore server

The ObjectStore server is the process that controls access to ObjectStore databases on a host and is responsible for the following:

- storage and retrieval of persistent data;
- handling concurrent access by multiple client applications;
- database recovery.

Client application

The ObjectStore client library is linked into each client application, allowing the client application to:

- map persistent objects to virtual addresses;
- allocate and deallocate storage for persistent objects;
- maintain a cache of recently used pages and the lock status of those pages;
- handle page faults on addresses that refer to persistent objects.

Client manager

The cache manager is a UNIX daemon/Windows service that runs on the same machine as the client application. Its function is to respond to server requests as a stand-in for the client application and manage the application's client cache, which exists to improve access to persistent objects. The client cache is the local buffer for data mapped, or

waiting to be mapped, into virtual memory. When a client application needs to access a persistent object, a page fault is generated in the following situations:

- the object is not in physical memory and not in the client cache;
- the object is in the client cache but has not yet been accessed;
- the object is in the client cache but has been previously accessed with different read/write permissions.

In these cases, the ObjectStore client requests the page from the server, copies it into the client cache, and resumes execution. If none of these conditions hold, the object in cache is available and the application just accesses it.

Ownership, locking, and the cache manager

To understand the function of the cache manager, we first have to understand ObjectStore's ownership and locking mechanisms. A client can request read or write permission for a page from the server. Read ownership can be granted to as many clients as request it, provided no client has write ownership, but there can be only one client with write ownership at any one time. When a client wants to read or write a page during a transaction it places a read or write lock on that page, thereby preventing any other client from receiving write permission for that page. A client must have read or write ownership to be able to place a read or write lock on a page. Once the transaction completes, the client releases the lock (although it can retain ownership).

Note the distinction between ownership and locks: ownership gives the client permission to read or update a page whereas a lock allows the client to actually read or update the page. With page ownership, a client can lock a page without communicating first with the server.

When a client requests permission to read a page and no other client has permission to update that page, the server can grant read ownership and the cache manager is not involved. However, the cache manager is involved when:

- a client requests read or write permission on a page and another client has write permission on that page;
- a client requests write permission on a page and at least one other client has read permission on that page.

In this case, the server sends a **callback message** to the cache manager associated with the client that has permission. This allows the client to concentrate on running the application and relieves it from having to listen for callback messages. Instead, the cache manager determines whether read or write permission can be released or whether the requesting client has to wait.

Virtual memory mapping architecture

One unique feature of ObjectStore is the way it handles persistence. ObjectStore stores a C++ object in the database on disk in its native format with all pointers intact (as opposed to swizzling them to OIDs as we discussed in Section 26.2.1). A full explanation of how

this process works is beyond the scope of this book and so we concentrate instead on a general overview of the mechanism.

The basic idea of the ObjectStore virtual memory mapping architecture is the same as for virtual memory management in operating systems. References to objects are realized by virtual memory addresses. If an object has to be dereferenced and the page the object resides on is already in main memory, there is no additional overhead in dereferencing this object and dereferencing is as fast as for any 'C' or C++ program. If the required page is not in main memory, a page fault occurs and the page is brought into the same virtual memory address it originally occupied. In this way, pointers to this object in other transferred objects are valid virtual memory pointers referring to their original target.

ObjectStore manages this process by reserving a range of unmapped virtual memory for persistent objects, thereby ensuring that this range will be used for no other purpose than database pages. When a program accesses its first object, ObjectStore transfers the page containing this object into virtual memory. When the program attempts to navigate from this initial object to another object using the second object's pointer, ObjectStore ensures that this pointer points to an unmapped portion of virtual memory. This results in the operating system raising a page fault, which ObjectStore traps and uses to bring the database page containing the second object into virtual memory.

When a program first attempts to update a page, another operating system exception is raised (a **write fault**). Again, ObjectStore traps this exception, transfers the page into virtual memory, if necessary, and changes the page's protection to read/write. The program then proceeds with the update. When the program wishes to save its updates, ObjectStore copies all pages that have been marked for update to the database and resets their protection back to read-only. When a database is closed, ObjectStore unmaps all its pages from virtual memory and unreserves the database's virtual memory range. With this approach, the programmer sees no difference between persistent and transient data.

27.3.2 Building an ObjectStore Application

Building a C++ ObjectStore application is slightly different from that described for the ODMG C++ language bindings in Section 27.2.5, as we discuss in this section. An ObjectStore application is built from a number of files:

- C++ source files that contain the main application code;
- C++ header files that contain the persistent classes;
- the necessary ObjectStore header files (for example, ostore.hh);
- a schema source file that defines the persistent classes for the schema generator.

Building an ObjectStore application requires the generation of the necessary schema information, that is, information about the classes the application stores in, or reads from, persistent memory. The schema source file is a C++ file containing a list of persistent classes and any reachable classes identified using the ObjectStore macro OS_MARK_SCHEMA_TYPE (a class is reachable if it is the base class or the class of a member of a persistent object). For example:

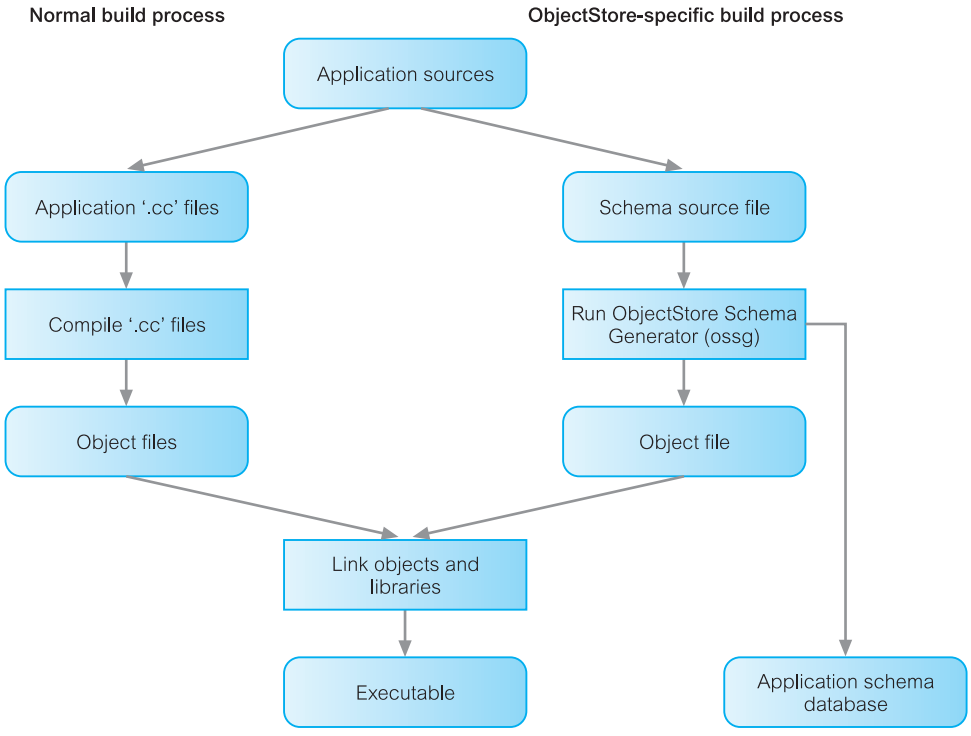


Figure 27.18 Building an ObjectStore application.

```
#include <ostore/ostore.hh>
#include <ostore/manschem.hh>
#include "myClasses.hh"                 /*defines persistent classes */
OS_MARK_SCHEMA_TYPE(Branch);            /*include Branch in schema */
OS_MARK_SCHEMA_TYPE(SalesStaff);        /* include SalesStaff in schema */
```

The ObjectStore schema generator (ossg) is run on this file to create two output files:

- an *application schema database* (for example, mySchema.adb), which contains type information about the objects the application can store persistently;
- an *application schema object file* (for example, mySchema.obj), which gets linked into the application.

The application is compiled in the normal way as is the output from the schema generator. The resulting object files are then linked to create the executable image, as illustrated in Figure 27.18.

ObjectStore databases

An ObjectStore database stores persistent objects and can be created using the os_database::create function. ObjectStore supports two types of database:

- *file database*, which is a native operating system file that contains an ObjectStore database;
- *rawfs (raw file system) database*, which is a private file system managed by the ObjectStore server, independent of the file system managed by the operating system.

An ObjectStore database is divided into clusters and segments. A *cluster* is the basic unit of storage allocation in an ObjectStore database. When a persistent object is created the storage is allocated from a cluster. Clusters are divided into *segments*. When a database is created, two segments are usually created:

- the *schema segment*, which holds the database roots and schema information about the objects stored in the database;
- the *default segment*, which stores entities created with the persistent version of the new operator.

Additional segments can be created using the function os_database::create_segment. Note that the schema segment cannot be accessed directly by the user application. Segments are allocated storage from a default cluster. When an application creates an object in persistent storage, it specifies the database to contain the object and the object is created in the default cluster of the default segment of this database. Alternatively, the application can specify a segment, in which case the object is created in the default cluster of the specified segment. Alternatively, the application can specify a cluster, in which case the object is created in the specified cluster.

27.3.3 Data Definition in ObjectStore

ObjectStore can handle persistence for objects created in the 'C', C++, and Java programming languages through separate class libraries, and there is a facility for objects created in one language to be accessed in the other. In this section we describe the C++ class library, which contains data members, member functions, and enumerators that provide access to database functionality.

ObjectStore uses C++ as a schema language so that everything in an ObjectStore database must be defined by a C++ class. In ObjectStore, persistence is orthogonal to type (see Section 26.3.2) and persistent object support is achieved through overloading the new operator, which allows dynamic allocation of persistent memory for any type of object. There is also a version of the C++ delete operator that can be used to delete persistent objects and free persistent memory. Once persistent memory has been allocated, pointers to this memory can be used in the same way as pointers to virtual memory is used. In fact, pointers to persistent memory always take the form of virtual memory pointers.

Figure 27.19 illustrates a possible set of ObjectStore C++ class declarations (in a '.h' *header file*) for part of the *DreamHome* database schema, concentrating on the Branch and SalesStaff classes and the relationship between them (Branch *Has* SalesStaff and SalesStaff *WorksAt* Branch). Much of the syntax in this schema will be familiar to readers with knowledge of C++. However, we discuss a few particular implementation details: creating persistent objects, relationships, and class extents.

Creating persistent objects by overloading the new operator

As we mentioned above, persistence is achieved by overloading the new operator. Figure 27.19 has two examples of the overloading of this operator in the constructors for the Branch and SalesStaff classes. For example, in the constructor for Branch, we have the statement:

```
branchNo = new(dreamhomeDB, os_typespec::get_char(), 4) char[4];
```

In this case, the new operator has three parameters:

- a pointer to an ObjectStore database (dreamhomeDB);
- a pointer to a type specification for the new object, which we have obtained by calling the overloaded method get_char of the os_typespec class (which we discuss next);
- the size of the object.

As usual, this version of the new operator returns a pointer to the newly allocated memory. Once an object has been created as persistent, ObjectStore will automatically retrieve it when a pointer to it is dereferenced. The examples given in Figure 27.19 are illustrative only; clearly in a complete implementation we would have to allocate space for all the attributes in Branch and SalesStaff rather than just the primary key attributes. Note, if we had omitted these parameters and used the standard version of the new operator, that is:

```
branchNo = new char[4];
```

then a transient object would have been created.

Using typespecs

Typespecs, instances of the class os_typespec, are used as arguments to the persistent version of the new operator to help maintain type safety when database roots are being manipulated (we discuss database roots in Section 27.3.3). A typespec represents a particular type, such as *char*, *int*, or *Branch**. ObjectStore provides some special functions for retrieving typespecs for various types. The first time such a function is called by a particular process, ObjectStore allocates the typespec and returns a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same os_typespec object is returned. In Figure 27.19, we have added members to the classes Branch and SalesStaff using a get_os_typespec member function:

```
static os_typespec *get_os_typespec();
```

The ObjectStore schema generator automatically supplies a body for this function, returning a pointer to a typespec for the class.

Creating relationships in ObjectStore

The relationship between Branch and SalesStaff is handled by declaring two data members that are the inverse of each other. With these bidirectional links, ObjectStore will automatically maintain referential integrity for this relationship. ObjectStore provides macros for defining relationships – Figure 27.19 uses two of these macros: os_relationship_1_m and os_relationship_m_1 (there are also macros called os_relationship_1_1 and os_relationship_m_m). These macros define access functions for setting and getting the

Figure 27.19
ObjectStore C++ class declarations for part of *DreamHome* database schema.

```
class SalesStaff;
extern os_Set<SalesStaff*> *salesStaffExtent;
extern os_database *dreamhomeDB;
enum PositionType {Manager, Supervisor, Assistant};
enum SexType {M, F};
struct Date {
   int year;
   int month;
   int day;
}
class Branch {                          // Define class for Branch
   char branchNo[4];
   struct {
        char* street;
        string* city;
        string* postcode} address;
os_relationship_m_1(Branch, Has, SalesStaff, WorksAt, os_Set<SalesStaff*>) Has;
Branch(char b[4]) {branchNo = new(dreamhomeDB, os_typespec::get_char(), 4) char[4];
                   strcpy(branchNo, b); }
// Provide a functional interface for creating the relationship – note this also sets up the inverse
// relationship WorksAt.
   void addStaff(SalesStaff *s) {Has.insert(s);}
   void removeStaff(SalesStaff *s) {Has.remove(s);}
   static os_typespec* get_os_typespec();
}
class Person {                          // Define class for Person
   struct {
        char* fName,
        char* lName} name;
}
class Staff: public Person {            // Define class for Staff that inherits from Person
   char staffNo[5];
   SexType sex;
   PositionType position;
   Date DOB;
   float salary;
   int getAge();
   void increaseSalary(float increment) {salary += increment; }
}
class SalesStaff : Staff {              // Define class for SalesStaff that inherits from Staff
   os_relationship_1_m(SalesStaff, WorksAt, Branch, Has, Branch*) WorksAt;
   SalesStaff(char s[5]) {staffNo = new(dreamhomeDB, os_typespec::get_char(), 5) char[5];
                          strcpy(staffNo, s);
                          salesStaffExtent->insert(this);}
~SalesStaff() {salesStaffExtent->remove(this);}
// Provide a functional interface for creating the relationship – note this also sets up the inverse
// relationship Has.
   void setBranch(Branch* b) {WorksAt.setvalue(b);}
   Branch* getBranch() {WorksAt.getvalue();}
   static os_typespec* get_os_typespec();
}
```

relationships. Each use of a relationship macro to define one side of a relationship must be paired with another relationship macro to define the other (inverse) side of the relationship. In each case, these macros take five parameters:

- **class** is the class that defines the data member being declared;
- **member** is the name of the member being declared;
- **inv_class** is the name of the class that defines the inverse member;
- **inv_member** is the name of the inverse member;
- **value_type** is the apparent value-type of the member being declared, which we discuss shortly.

To instantiate relationship functions, there is an associated set of relationship 'body' macros that take the same first four parameters (which must be invoked from a source file). For example, to match the two relationship macros in Figure 27.19, we need the following two statements:

```
os_rel_m_1 _body(Branch, Has, SalesStaff, WorksAt);
os_rel_1_m _body(SalesStaff, WorksAt, Branch, Has);
```

We have also provided a functional interface to these relationships through the methods addStaff and removeStaff in Branch and setBranch and getBranch in Staff. Note also the transparency of the bidirectional relationships; for example, when we invoke the addStaff method to specify that this branch (b1, say) *Has* the given member of staff (s1 say), the inverse relationship *WorksAt* is also set up (that is, s1 *WorksAt* b1).

Creating extents in ObjectStore

In Figure 27.15 we specified an extent for the SalesStaff class using the ODMG keyword **extent**. On the second line of Figure 27.19 we have also specified an extent for SalesStaff using the ObjectStore collection type os_Set. In the SalesStaff constructor, we have used the insert method to insert the object into the class extent and in the destructor we have used the remove method to delete the object from the class extent.

Data Manipulation in ObjectStore 27.3.4

In this section we briefly discuss the manipulation of objects in an ObjectStore database. The following operations must be performed before persistent memory can be accessed:

- a database must be created or opened;
- a transaction must be started;
- a database root must be retrieved or created.

Roots and entry point objects

As we mentioned in Section 27.2.2, a database root provides a way to give an object a persistent name, thereby allowing the object to serve as an initial *entry point* into the

Figure 27.20
Creating persistent objects and relationships in ObjectStore.

```
os_Set<SalesStaff*> *salesStaffExtent = 0;
main() {
// Initialize ObjectStore and initialize use of collections
   objectstore::initialize(); os_collection::initialize();
   os_typespec *WorksAtType = Branch::get_os_typespec();
   os_typespec *salesStaffExtentType = os_Set<SalesStaff*>::get_os_typespec();
// Open the DreamHome database
   os_database *db1 = os_database::open("dreamhomeDB");
// Begin a transaction
   OS_BEGIN_TXN(tx1, 0, os_transaction::update)
// Create the SalesStaff extent in this database and then create a named root
            salesStaffExtent = &os_Set<SalesStaff*>::create(db1);
            db1->create_root("salesStaffExtent_Root")->set_value(salesStaffExtent, salesStaffExtentType);
// Create branch B003 with two staff, SG37 and SG14
            Branch* b1("B003"); SalesStaff* s1("SG37"), s2("SG14");
// Create a root for B003 and set the two staff as working at this branch
            db1->create_root("Branch3_Root")->set_value(b1, WorksAtType);
            b1->addStaff(s1); b1->addStaff(s2);
// End the transaction and close the database
OS_END_TXN(tx1)
db1->close();
delete db1;
objectstore::shutdown();
}
```

database. From there, any object related to it can be retrieved using *navigation* (that is, following data member pointers) or by a *query* (that is, selecting all elements of a given collection that satisfy a specified predicate). Figure 27.20 illustrates a number of these points:

- Opening the database using the open method of the database class os_database.
- Starting and stopping a transaction using the macros OS_BEGIN_TXN and OS_END_TXN (the first parameter is an identifier, tx1, that simply serves as a label for the transaction).
- The creation of an extent for SalesStaff using the create method of the collection class os_Set.
- The creation of two named roots (one for the SalesStaff extent and one corresponding to branch B003) using the create_root method of the database class os_database. This method returns a pointer to the new root (of type os_database_root), which is then used to specify the name to be associated with the root using the set_value method.
- The creation of a Branch instance representing branch B003 followed by two SalesStaff instances, SG37 and SG14, which are then added as staff at B003 using the addStaff method of the Branch class.

```
os_Set<SalesStaff*> *salesStaffExtent = 0;
main() {
   Branch* aBranch;
   SalesStaff* p;
// Initialize ObjectStore and initialize use of collections
   objectstore::initialize(); os_collection::initialize();
   os_typespec *WorksAtType = Branch::get_os_typespec();
   os_typespec *salesStaffExtentType = os_Set<SalesStaff*>::get_os_typespec();
// Open the database and start a transaction
   os_database *db1 = os_database::open("dreamhomeDB");
   OS_BEGIN_TXN(tx1, 0, os_transaction::update)
// Query 1. Find named Branch3 root and use a cursor to find all staff at this branch
        aBranch = (Branch*)(db1->find_root("Branch3_Root")->get_value(WorksAtType);
        cout << "Retrieval of branch B003 root:" << aBranch->branchNo << "\n";
// Query 2. Now find all staff at this branch
        os_Cursor<SalesStaff*> c(aBranch->Has);
        count << "Staff associated with branch B003: \n"
        for (p = c.first(); c.more(); p = c.next())
            cout << p->staffNo << "\n";
// Query 3. Find named SalesStaffExtent root and carry out a query on this extent
        salesStaffExtent = (os_Set<SalesStaff*>*)
                (db1->find_root("salesStaffExtent_Root")->get_value(salesStaffExtentType);
        aSalesPerson = salesStaffExtent->query_pick("SalesStaff*", "!strcmp(staffNo, \"SG37\")", db1);
        cout << "Retrieval of specific member of sales staff: " << aSalesPerson.staffNo << "\n";
// Query 4. Carry out another query on this extent to find highly paid staff (use cursor to traverse set returned)
        os_Set<SalesStaff*> &highlyPaidStaff =
                salesStaffExtent->query("SalesStaff*", "salary > 30000", db1);
        cout << "Retrieval of highly paid staff:  \n";
        os_Cursor<SalesStaff*> c(highlyPaidStaff);
        for (p = c.first(); c.more(); p = c.next())
            cout << p->staffNo << "\n";
   OS_END_TXN(tx1)
   db1->close();
   delete db1;
   objectstore::shutdown();
}
```

Access based on a named root

Iteration of collections using cursors

Object lookup

Retrieval of collection of objects

Figure 27.21 Querying in ObjectStore.

Queries

ObjectStore provides a number of ways to retrieve objects from the database covering both navigational and associative access. Figure 27.21 illustrates some methods for retrieving objects:

- *Access based on a named root* In the previous example, we created a named root for branch B003 and we can now use this root to retrieve the branch object B003 and display its branch number, branchNo. This is achieved using the find_root and get_value methods, analogous with the create_root and set_value methods used in Figure 27.20.

- *Iteration of collections using cursors* Having found the branch B003 object, we can now use the relationship *Has* to iterate over the sales staff assigned to that branch (the *Has* relationship was defined in Figure 27.19 as a collection, os_Set). The ObjectStore collection facility provides a number of classes to help navigate within a collection. In this example, we have used the **cursor mechanism**, which is used to designate a position within a collection (similar to the SQL cursor mechanism discussed in Appendix E.1.4). Cursors can be used to traverse collections, as well as to retrieve, insert, remove, and replace elements. To find the sales staff at branch B003, we have created an instance of the parameterized template class os_Cursor, c, using the collection of sales staff that has been defined through the *Has* relationship, in this case aBranch->Has. We can then iterate over this collection using the cursor methods first (which moves to the first element in the set), next (which moves to the next element in the set), and more (which determines whether there are any other elements in the set).

These first two examples are based on navigational access, whereas the remaining two examples illustrate associative access.

- *Lookup of a single object based on the value of one or more data members* ObjectStore supports associative access to persistent objects. We illustrate the use of this mechanism using the SalesStaff extent and, as a first example, we retrieve one element of this extent using the query_pick method, which takes three parameters:
  - a string indicating the element type of the collection being queried (in this case SalesStaff*);
  - a string indicating the condition that elements must satisfy in order to be selected by the query (in this case the element where the staffNo data member is SG37);
  - a pointer to the database containing the collection being queried (in this case db1).
- *Retrieval of a collection of objects based on the value of one or more data members* To extend the previous example, we use the query method to return a number of elements in the collection that satisfy a condition (in this case, those staff with a salary greater than £30,000). This query returns another collection and we again use a cursor to iterate over the elements of the collection and display the staff number, staffNo.

In this section we have only touched on the features of the ObjectStore OODBMS. The interested reader is referred to the ObjectStore system documentation for further information.

Chapter Summary

- The **Object Management Group** (OMG) is an international non-profit-making industry consortium founded in 1989 to address the issues of object standards. The primary aims of the OMG are promotion of the object-oriented approach to software engineering and the development of standards in which the location, environment, language, and other characteristics of objects are completely transparent to other objects.
- In 1990, the OMG first published its **Object Management Architecture** (OMA) Guide document. This guide specified a single terminology for object-oriented languages, systems, databases, and application frameworks;

an abstract framework for object-oriented systems; a set of technical and architectural goals; and a reference model for distributed applications using object-oriented techniques. Four areas of standardization were identified for the reference model: the Object Model (OM), the Object Request Broker (ORB), the Object Services, and the Common Facilities.

- CORBA defines the architecture of ORB-based environments. This architecture is the basis of any OMG component, defining the parts that form the ORB and its associated structures. Using GIOP or IIOP, a CORBA-based program can interoperate with another CORBA-based program across a variety of vendors, platforms, operating systems, programming languages, and networks. Some of the elements of CORBA are an implementation-neutral **Interface Definition Language** (IDL), a **type model**, an **Interface Repository**, methods for getting the interfaces and specifications of objects, and methods for transforming OIDs to and from strings.
- The OMG has also developed a number of other specifications including the **UML** (Unified Modeling Language), which provides a common language for describing software models; **MOF** (Meta-Object Facility), which defines a common, abstract language for the specification of metamodels (CORBA, UML, and CWM are all MOF-compliant metamodels); **XMI** (XML Metadata Interchange), which maps MOF to XML; and **CWM** (Common Warehouse Metamodel), which defines a metamodel for metadata that is commonly found in data warehousing and business intelligence domains.
- The OMG has also introduced the **Model-Driven Architecture** (**MDA**) as an approach to system specification and interoperability building upon the above four modeling specifications. It is based on the premise that systems should be specified independently of all hardware and software details. Thus, while the software and hardware may change over time, the specification will still be applicable. Importantly, MDA addresses the complete system lifecycle, from analysis and design to implementation, testing, component assembly, and deployment.
- Several important vendors formed the **Object Data Management Group** (ODMG) to define standards for OODBMSs. The ODMG produced an Object Model that specifies a standard model for the semantics of database objects. The model is important because it determines the built-in semantics that the OODBMS understands and can enforce. The design of class libraries and applications that use these semantics should be portable across the various OODBMSs that support the Object Model.
- The major components of the ODMG architecture for an OODBMS are: an Object Model (OM), an Object Definition Language (ODL), an Object Query Language (OQL), and C++, Java, and Smalltalk language bindings.
- The ODMG OM is a superset of the OMG OM, which enables both designs and implementations to be ported between compliant systems. The basic modeling primitives in the model are the **object** and the **literal**. Only an object has a unique identifier. Objects and literals can be categorized into **types**. All objects and literals of a given type exhibit common behavior and state. Behavior is defined by a set of **operations** that can be performed on or by the object. State is defined by the values an object carries for a set of **properties**. A property may be either an **attribute** of the object or a **relationship** between the object and one or more other objects.
- The **Object Definition Language** (ODL) is a language for defining the specifications of object types for ODMG-compliant systems, equivalent to the Data Definition Language (DDL) of traditional DBMSs. The ODL defines the attributes and relationships of types and specifies the signature of the operations, but it does not address the implementation of signatures.
- The **Object Query Language** (OQL) provides declarative access to the object database using an SQL-like syntax. It does not provide explicit update operators, but leaves this to the operations defined on object types. An OQL query is a function that delivers an object whose type may be inferred from the operator contributing to the query expression. OQL can be used for both associative and navigational access.

Review Questions

27.1 Discuss the main concepts of the ODMG Object Model. Give an example to illustrate each of the concepts.

27.2 What is the function of the ODMG Object Definition Language?

27.3 What is the function of the ODMG Object Manipulation Language?

27.4 How does the ODMG GROUP BY clause differ from the SQL GROUP BY clause? Give an example to illustrate your answer.

27.5 How does the ODMG aggregate functions differ from the SQL aggregate functions? Give an example to illustrate your answer.

27.6 What is the function of the ODMG Object Interchange Format?

27.7 Briefly discuss how the ODMG C++ language binding works.

Exercises

27.8 Map the object-oriented database design for the Hotel case study produced in Exercise 26.14 and then show how the following queries would be written in OQL:

(a) List all hotels.
(b) List all single rooms with a price below £20 per night.
(c) List the names and cities of all guests.
(d) List the price and type of all rooms at the Grosvenor Hotel.
(e) List all guests currently staying at the Grosvenor Hotel.
(f) List the details of all rooms at the Grosvenor Hotel, including the name of the guest staying in the room, if the room is occupied.
(g) List the guest details (guestNo, guestName, and guestAddress) of all guests staying at the Grosvenor Hotel.

Compare the OQL answers with the equivalent relational algebra and relational calculus expressions of Exercise 4.12.

27.9 Map the object-oriented database design for the *DreamHome* case study produced in Exercise 26.15 to the ODMG ODL.

27.10 Map the object-oriented database design for the *University Accommodation Office* case study produced in Exercise 26.16 to the ODMG ODL.

27.11 Map the object-oriented database design for the *EasyDrive School of Motoring* case study produced in Exercise 26.17 to the ODMG ODL.

27.12 Map the object-oriented database design for the *Wellmeadows* case study produced in Exercise 26.18 to the ODMG ODL.

Chapter

28 Object-Relational DBMSs

Chapter Objectives

In this chapter you will learn:

- How the relational model has been extended to support advanced database applications.
- The features proposed in the third-generation database system manifestos presented by CADF, and Darwen and Date.
- The extensions to the relational data model that have been introduced to Postgres.
- The object-oriented features in the new SQL standard, SQL:2003, including:
  - row types;
  - user-defined types and user-defined routines;
  - polymorphism;
  - inheritance;
  - reference types and object identity;
  - collection types (ARRAYs, MULTISETs, SETs, and LISTs);
  - extensions to the SQL language to make it computationally complete;
  - triggers;
  - support for large objects: Binary Large Objects (BLOBs) and Character Large Objects (CLOBs);
  - recursion.
- Extensions required to relational query processing and query optimization to support advanced queries.
- Some object-oriented extensions to Oracle.
- How OODBMSs and ORDBMSs compare in terms of data modeling, data access, and data sharing.

In Chapters 25 to 27 we examined some of the background concepts of object-orientation and Object-Oriented Database Management Systems (OODBMSs). In Chapter 25 we also looked at the types of advanced database application that are emerging and the weaknesses of current RDBMSs that make them unsuitable for these types of application. In

Chapters 26 and 27 we discussed the OODBMS in detail and the mechanisms that make it more suitable for these advanced applications. In response to the weaknesses of relational systems, and in defense of the potential threat posed by the rise of the OODBMS, the RDBMS community has extended the RDBMS with object-oriented features, giving rise to the **Object-Relational DBMS** (ORDBMS). In this chapter we examine some of these extensions and how they help overcome many of the weaknesses cited in Section 25.2. We also examine some of the problems that are introduced by these new extensions in overcoming the weaknesses.

Structure of this Chapter

In Section 28.1 we examine the background to the ORDBMS and the types of application that they may be suited to. In Section 28.2 we examine two third-generation manifestos based on the relational data model that provide slightly different insights into what the next generation of DBMS should look like. In Section 28.3 we investigate an early extended RDBMS, followed in Section 28.4 by a detailed review of the main features of the SQL:1999 standard released in 1999 and the SQL:2003 standard released in the second half of 2003. In Section 28.5 we discuss some of the functionality that an ORDBMS will typically require that is not covered by SQL. In Section 28.6 we examine some of the object-oriented extensions that have been added to Oracle, a commercial ORDBMS. Finally, in Section 28.7 we provide a summary of the distinctions between the ORDBMS and the OODBMS.

To benefit fully from this chapter, the reader needs to be familiar with the contents of Chapter 25. The examples in this chapter are once again drawn from the *DreamHome* case study documented in Section 10.4 and Appendix A.

28.1 Introduction to Object-Relational Database Systems

Relational DBMSs are currently the dominant database technology with estimated sales of between US$6 billion and US$10 billion per year (US$25 billion with tools sales included). The OODBMS, which we discussed in Chapters 26 and 27, started in the engineering and design domains, and has also become the favored system for financial and telecommunications applications. Although the OODBMS market is still small, the OODBMS continues to find new application areas, such as the Web (which we discuss in detail in Chapter 29). Some industry analysts expect the market for the OODBMS to grow at a rate faster than the total database market. However, their sales are unlikely to overtake those of relational systems because of the wealth of businesses that find RDBMSs acceptable, and because businesses have invested so much money and resources in their development that change is prohibitive.

Until recently, the choice of DBMS seemed to be between the relational DBMS and the object-oriented DBMS. However, many vendors of RDBMS products are conscious of the threat and promise of the OODBMS. They agree that traditional relational DBMSs are not suited to the advanced applications discussed in Section 25.1, and that added functionality is required. However, they reject the claim that extended RDBMSs will not provide sufficient functionality or will be too slow to cope adequately with the new complexity.

If we examine the advanced database applications that are emerging, we find they make extensive use of many object-oriented features such as a user-extensible type system, encapsulation, inheritance, polymorphism, dynamic binding of methods, complex objects including non-first normal form objects, and object identity. The most obvious way to remedy the shortcomings of the relational model is to extend the model with these types of feature. This is the approach that has been taken by many extended relational DBMSs, although each has implemented different combinations of features. Thus, there is no single extended relational model; rather, there are a variety of these models, whose characteristics depend upon the way and the degree to which extensions were made. However, all the models do share the same basic relational tables and query language, all incorporate some concept of 'object', and some have the ability to store methods (or procedures or triggers) as well as data in the database.

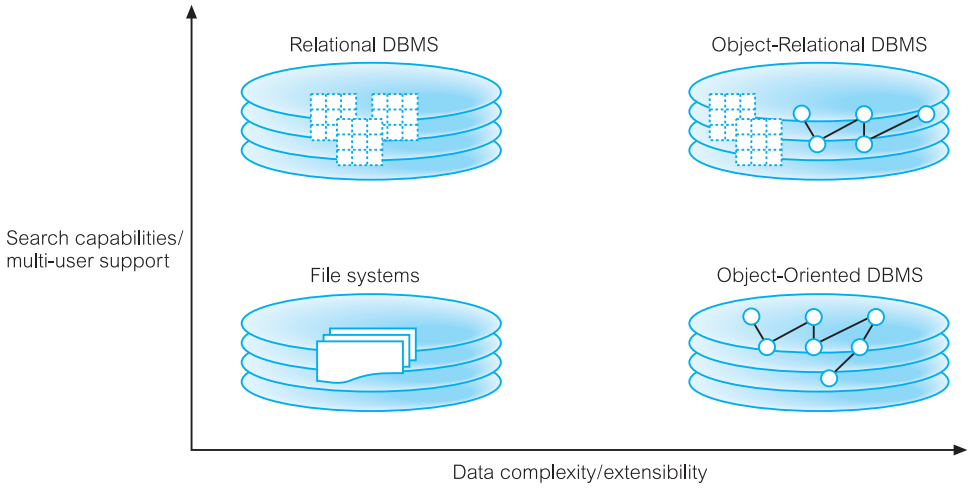
Various terms have been used for systems that have extended the relational data model. The original term that was used to describe such systems was the *Extended Relational DBMS* (ERDBMS). However, in recent years the more descriptive term *Object-Relational DBMS* has been used to indicate that the system incorporates some notion of '*object*', and the term *Universal Server* or *Universal DBMS* (UDBMS) has also been used. In this chapter we use the term Object-Relational DBMS (ORDBMS). Three of the leading RDBMS vendors – Oracle, Microsoft, and IBM – have all extended their systems into ORDBMSs, although the functionality provided by each is slightly different. The concept of the ORDBMS, as a hybrid of the RDBMS and the OODBMS, is very appealing, preserving the wealth of knowledge and experience that has been acquired with the RDBMS. So much so, that some analysts predict the ORDBMS will have a 50% larger share of the market than the RDBMS.

As might be expected, the standards activity in this area is based on extensions to the SQL standard. The national standards bodies have been working on object extensions to SQL since 1991. These extensions have become part of the SQL standard, with releases in 1999, referred to as SQL:1999, and 2003, referred to as SQL:2003. These releases of the SQL standard are an ongoing attempt to standardize extensions to the relational model and query language. We discuss the object extensions to SQL in some detail in Section 28.4. In this book, we generally use the term SQL:2003 to refer to both the 1999 and 2003 releases of the standard.

Stonebraker's view

Stonebraker (1996) has proposed a four-quadrant view of the database world, as illustrated in Figure 28.1. In the lower-left quadrant are those applications that process simple data and have no requirements for querying the data. These types of application, for example standard text processing packages such as Word, WordPerfect, and Framemaker, can use the underlying operating system to obtain the essential DBMS functionality of persistence.

Figure 28.1



In the lower-right quadrant are those applications that process complex data but again have no significant requirements for querying the data. For these types of application, for example computer-aided design packages, an OODBMS may be an appropriate choice of DBMS. In the top-left quadrant are those applications that process simple data and also have requirements for complex querying. Many traditional business applications fall into this quadrant and an RDBMS may be the most appropriate DBMS. Finally, in the top-right quadrant are those applications that process complex data and have complex querying requirements. This represents many of the advanced database applications that we examined in Section 25.1 and for these applications an ORDBMS may be the most appropriate choice of DBMS.

Although interesting, this is a very simplistic classification and unfortunately many database applications are not so easily compartmentalized. Further, with the introduction of the ODMG data model and query language, which we discussed in Section 27.2, and the addition of object-oriented data management features to SQL, the distinction between the ORDBMS and OODBMS is becoming less clear.

Advantages of ORDBMSs

Apart from the advantages of resolving many of the weaknesses cited in Section 25.2, the main advantages of extending the relational data model come from *reuse* and *sharing*. Reuse comes from the ability to extend the DBMS server to perform standard functionality centrally, rather than have it coded in each application. For example, applications may require spatial data types that represent points, lines, and polygons, with associated functions that calculate the distance between two points, the distance between a point and a line, whether a point is contained within a polygon, and whether two polygonal regions overlap, among others. If we can embed this functionality in the server, it saves having to define it in each application that needs it, and consequently allows the functionality to be shared by all applications. These advantages also give rise to increased productivity both for the developer and for the end-user.

Another obvious advantage is that the extended relational approach preserves the significant body of knowledge and experience that has gone into developing relational applications. This is a significant advantage, as many organizations would find it prohibitively expensive to change. If the new functionality is designed appropriately, this approach should allow organizations to take advantage of the new extensions in an evolutionary way without losing the benefits of current database features and functions. Thus, an ORDBMS could be introduced in an integrative fashion, as proof-of-concept projects. The SQL:2003 standard is designed to be upwardly compatible with the SQL2 standard, and so any ORDBMS that complies with SQL:2003 should provide this capability.

Disadvantages of ORDBMSs

The ORDBMS approach has the obvious disadvantages of complexity and associated increased costs. Further, there are the proponents of the relational approach who believe the essential simplicity and purity of the relational model are lost with these types of extension. There are also those who believe that the RDBMS is being extended for what will be a minority of applications that do not achieve optimal performance with current relational technology.

In addition, object-oriented purists are not attracted by these extensions either. They argue that the terminology of object-relational systems is revealing. Instead of discussing object models, terms like 'user-defined data types' are used. The terminology of object-orientation abounds with terms like 'abstract types', 'class hierarchies', and 'object models'. However, ORDBMS vendors are attempting to portray object models as extensions to the relational model with some additional complexities. This potentially misses the point of object-orientation, highlighting the large semantic gap between these two technologies. Object applications are simply not as data-centric as relational-based ones. Object-oriented models and programs deeply combine relationships and encapsulated objects to more closely mirror the 'real world'. This defines broader sets of relationships than those expressed in SQL, and involves functional programs interspersed in the object definitions. In fact, objects are fundamentally not extensions of data, but a completely different concept with far greater power to express 'real-world' relationships and behaviors.

In Chapter 5 we noted that the objectives of a database language included having the capability to be used with minimal user effort, and having a command structure and syntax that must be relatively easy to learn. The initial SQL standard, released in 1989, appeared to satisfy these objectives. The release in 1992 increased in size from 120 pages to approximately 600 pages, and it is more questionable whether it satisfied these objectives. Unfortunately, the size of the SQL:2003 standard is even more daunting, and it would seem that these two objectives are no longer being fulfilled or even being considered by the standards bodies.

28.2 The Third-Generation Database Manifestos

The success of relational systems in the 1990s is evident. However, there is significant dispute regarding the next generation of DBMSs. The traditionalists believe that it is sufficient to extend the relational model with additional capabilities. On the one hand, one

influential group has published the *Object-Oriented Database System Manifesto* based on the object-oriented paradigm, which we presented in Section 26.1.4 (Atkinson *et al.*, 1989). On the other hand, the Committee for Advanced DBMS Function (CADF) has published the *Third-Generation Database System Manifesto* which defines a number of principles that a DBMS ought to meet (Stonebraker *et al.*, 1990). More recently, Darwen and Date (1995, 2000) have published the *Third Manifesto* in defense of the relational data model. In this section we examine both these manifestos.

28.2.1 The *Third-Generation Database System Manifesto*

The manifesto published by the CADF proposes the following features for a third-generation database system:

(1) A third-generation DBMS must have a rich type system.

(2) Inheritance is a good idea.

(3) Functions, including database procedures and methods and encapsulation, are a good idea.

(4) Unique identifiers for records should be assigned by the DBMS only if a user-defined primary key is not available.

(5) Rules (triggers, constraints) will become a major feature in future systems. They should not be associated with a specific function or collection.

(6) Essentially all programmatic access to a database should be through a non-procedural, high-level access language.

(7) There should be at least two ways to specify collections, one using enumeration of members and one using the query language to specify membership.

(8) Updateable views are essential.

(9) Performance indicators have almost nothing to do with data models and must not appear in them.

(10) Third-generation DBMSs must be accessible from multiple high-level languages.

(11) Persistent forms of a high-level language, for a variety of high-level languages, are a good idea. They will all be supported on top of a single DBMS by compiler extensions and a complex runtime system.

(12) For better or worse, SQL is 'intergalactic dataspeak'.

(13) Queries and their resulting answers should be the lowest level of communication between a client and a server.

28.2.2 *The Third Manifesto*

The Third Manifesto by Darwen and Date (1995, 2000) attempts to defend the relational data model as described in the authors' 1992 book (Date and Darwen, 1992). It is acknowledged that certain object-oriented features are desirable, but the authors believe these features to be orthogonal to the relational model, so that 'the relational model needs

no extension, no correction, no subsumption, and, above all, no perversion'. However, SQL is *unequivocally rejected* as a *perversion* of the model and instead a language called **D** is proposed. Instead, it is suggested that a frontend layer is furnished to D that allows SQL to be used, thus providing a migration path for existing SQL users. The manifesto proposes that D be subject to:

(1) prescriptions that arise from the relational model, called *RM Prescriptions*;

(2) prescriptions that do not arise from the relational model, called Other Orthogonal (OO) Prescriptions (*OO Prescriptions*);

(3) proscriptions that arise from the relational model, called *RM Proscriptions*;

(4) proscriptions that do not arise from the relational model, called *OO Proscriptions*.

In addition, the manifesto lists a number of very strong suggestions based on the relational model and some other orthogonal very strong suggestions. The proposals are listed in Table 28.1.

The primary object in the proposal is the **domain**, defined as *a named set of encapsulated values, of arbitrary complexity*, equivalent to a data type or object class. Domain values are referred to generically as scalars, which can be manipulated only by means of operators defined for the domain. The language D comes with some built-in domains, such as the domain of truth values with the normal boolean operators (AND, OR, NOT, and so on). The equals (=) comparison operator is defined for every domain, returning the boolean value TRUE if and only if the two members of the domain are the same. Both single and multiple inheritance on domains are proposed.

Relations, tuples, and tuple headings have their normal meaning with the introduction of RELATION and TUPLE type constructors for these objects. In addition, the following variables are defined:

- *Scalar variable of type V* Variable whose permitted values are scalars from a specified domain V.
- *Tuple variable of type H* Variable whose permitted values are tuples with a specified tuple heading H.
- *Relation variable (relvar) of type H* Variable whose permitted values are relations with a specified relation heading H.
- *Database variable (dbvar)* A named set of relvars. Every dbvar is subject to a set of named integrity constraints and has an associated self-describing catalog.

A transaction is restricted to interacting with only one dbvar, but can dynamically add/remove relvars from that dbvar. Nested transactions should be supported. It is further proposed that the language D should:

- Represent the relational algebra '*without excessive circumlocution*'.
- Provide operators to create/destroy named functions, whose value is a relation defined by means of a specified relational expression.
- Support the comparison operators:
  - (= and ≠) for tuples;
  - (=, ≠, 'is a subset of', ∈ for testing membership of a tuple in a relation) for relations.
- Be constructed according to well-established principles of good language design.

Table 28.1 *Third Manifesto* proposals.

RM prescriptions

(1) Scalar types
(2) Scalar values are typed
(3) Scalar operators
(4) Actual *vs* possible representation
(5) Expose possible representations
(6) Type generator TUPLE
(7) Type generator RELATION
(8) Equality
(9) Tuples
(10) Relations
(11) Scalar variables
(12) Tuple variables
(13) Relation variables (relvars)
(14) Base *vs* virtual relvars
(15) Candidate keys
(16) Databases
(17) Transactions
(18) Relational algebra
(19) Relvar names, relation selectors, and recursion
(20) Relation-valued operators
(21) Assignment
(22) Comparisons
(23) Integrity constraints
(24) Relation and database predicates
(25) Catalog
(26) Language design

RM proscriptions

(1) No attribute ordering
(2) No tuple ordering
(3) No duplicate tuples
(4) No nulls
(5) No nullological mistakes[a]
(6) No internal-level constructs
(7) No tuple-level operations
(8) No composite columns
(9) No domain check override
(10) Not SQL

OO prescriptions

(1) Compile-time type-checking
(2) Single inheritance (conditional)
(3) Multiple inheritance (conditional)
(4) Computational completeness
(5) Explicit transactions boundaries
(6) Nested transactions
(7) Aggregates and empty sets

OO proscriptions

(1) Relvars are not domains
(2) No object IDs

RM very strong suggestions

(1) System keys
(2) Foreign keys
(3) Candidate key inference
(4) Transition constraints
(5) Quota queries (for example, 'find three youngest staff')
(6) Generalized transitive closure
(7) Tuple and relation parameters
(8) Special ('default') values
(9) SQL migration

OO very strong suggestions

(1) Type inheritance
(2) Types and operators unbundled
(3) Collection type generators
(4) Conversion to/from relations
(5) Single-level store

[a] Darwen defines nullology as 'the study of nothing at all', meaning the study of the empty set. Sets are an important aspect of relational theory, and correct handling of the empty set is seen as fundamental to relational theory.

Postgres – An Early ORDBMS 28.3

In this section we examine an early Object-Relational DBMS, Postgres ('Post INGRES'). The objective of this section is to provide some insight into how some researchers have approached extending relational systems. However, it is expected that many mainstream ORDBMSs will conform to SQL:2003 (at least to some degree). Postgres is a research system from the designers of INGRES that attempts to extend the relational model with abstract data types, procedures, and rules. Postgres had an influence on the development of the object management extensions to the commercial product INGRES. One of its principal designers, Mike Stonebraker, subsequently went on to design the Illustra ORDBMS.

Objectives of Postgres 28.3.1

Postgres is a research database system designed to be a potential successor to the INGRES RDBMS (Stonebraker and Rowe, 1986). The stated objectives of the project were:

(1) to provide better support for complex objects;
(2) to provide user extensibility for data types, operators, and access methods;
(3) to provide active database facilities (alerters and triggers) and inferencing support;
(4) to simplify the DBMS code for crash recovery;
(5) to produce a design that can take advantage of optical disks, multiple-processor workstations, and custom-designed VLSI (Very Large Scale Integration) chips;
(6) to make as few changes as possible (preferably none) to the relational model.

Postgres extended the relational model to include the following mechanisms:

- abstract data types;
- data of type 'procedure';
- rules.

These mechanisms are used to support a variety of semantic and object-oriented data modeling constructs including aggregation, generalization, complex objects with shared subobjects, and attributes that reference tuples in other relations.

Abstract Data Types 28.3.2

An attribute type in a relation can be atomic or structured. Postgres provides a set of predefined atomic types: **int2**, **int4**, **float4**, **float8**, **bool**, **char**, and **date**. Users can add new atomic types and structured types. All data types are defined as abstract data types (ADTs). An ADT definition includes a type name, its length in bytes, procedures for converting a value from internal to external representation (and *vice versa*), and a default value. For example, the type **int4** is internally defined as:

DEFINE TYPE int4 **IS** (InternalLength = 4, InputProc = CharToInt4,
OutputProc = Int4ToChar, Default = "0")

The conversion procedures CharToInt4 and Int4ToChar are implemented in some high-level programming language such as 'C' and made known to the system using a **DEFINE PROCEDURE** command. An operator on ADTs is defined by specifying the number and type of operand, the return type, the precedence and associativity of the operator, and the procedure that implements it. The operator definition can also specify procedures to be called, for example, to sort the relation if a sort–merge strategy is selected to implement the query (Sort), and to negate the operator in a query predicate (Negator). For example, we could define an operator '+' to add two integers together as follows:

DEFINE OPERATOR "+" (int4, int4) **RETURNS** int4
IS (Proc = Plus, Precedence = 5, Associativity = "left")

Again, the procedure Plus that implements the operator '+' would be programmed in a high-level language. Users can define their own atomic types in a similar way.

Structured types are defined using type constructors for arrays and procedures. A variable-length or fixed-length array is defined using an **array constructor**. For example, char[25] defines an array of characters of fixed length 25. Omitting the size makes the array variable-length. The **procedure constructor** allows values of type 'procedure' in an attribute, where a procedure is a series of commands written in Postquel, the query language of Postgres (the corresponding type is called the **postquel** data type).

28.3.3 Relations and Inheritance

A relation in Postgres is declared using the following command:

CREATE TableName (columnName1 = type1, columnName2 = type2, . . .)
[**KEY**(listOfColumnNames)]
[**INHERITS**(listOfTableNames)]

A relation inherits all attributes from its parent(s) unless an attribute is overridden in the definition. Multiple inheritance is supported, however, if the same attribute can be inherited from more than one parent and the attribute types are different, the declaration is disallowed. Key specifications are also inherited. For example, to create an entity Staff that inherits the attributes of Person, we would write:

CREATE Person (fName = char[15], lName = char[15], sex = char, dateOfBirth = date)
KEY(lName, dateOfBirth)
CREATE Staff (staffNo = char[5], position = char[10], salary = float4,
branchNo = char[4], manager = postquel)
INHERITS(Person)

The relation Staff includes the attributes declared explicitly together with the attributes declared for Person. The key is the (inherited) key of Person. The manager attribute is defined as type *postquel* to indicate that it is a Postquel query. A tuple is added to the Staff relation using the **APPEND** command:

```
APPEND Staff (staffNo = "SG37", fName = "Ann", lName = "Beech", sex = "F",
              dateOfBirth = "10-Nov-60", position = "Assistant", salary = 12000,
              branchNo = "B003", manager = "RETRIEVE (s.staffNo) FROM s
              IN Staff WHERE position = 'Manager' AND branchNo = 'B003'")
```

A query that references the manager attribute returns the string that contains the Postquel command, which in general may be a relation as opposed to a single value. Postgres provides two ways to access the manager attribute. The first uses a nested dot notation to implicitly execute a query:

```
RETRIEVE (s.staffNo, s.lName, s.manager.staffNo) FROM s IN Staff
```

This query lists each member of staff's number, name, and associated manager's staff number. The result of the query in manager is implicitly joined with the tuple specified by the rest of the retrieve list. The second way to execute the query is to use the **EXECUTE** command:

```
EXECUTE (s.staffNo, s.lName, s.manager.staffNo) FROM s IN Staff
```

Parameterized procedure types can be used where the query parameters can be taken from other attributes in the tuple. The $ sign is used to refer to the tuple in which the query is stored. For example, we could redefine the above query using a parameterized procedure type:

```
DEFINE TYPE Manager IS
  RETRIEVE (staffNumber = s.staffNo) FROM s IN Staff WHERE
                position = "Manager" AND branchNo = $.branchNo
```

and use this new type in the table creation:

```
CREATE Staff(staffNo = char[5], position = char[10], salary = float4,
             branchNo = char[4], manager = Manager)
INHERITS(Person)
```

The query to retrieve staff details would now become:

```
RETRIEVE (s.staffNo, s.lName, s.manager.staffNumber) FROM s IN Staff
```

The ADT mechanism of Postgres is limited in comparison with OODBMSs. In Postgres, objects are composed from ADTs, whereas in an OODBMS all objects are treated as ADTs. This does not fully satisfy the concept of encapsulation. Furthermore, there is no inheritance mechanism associated with ADTs, only tables.

28.3.4 Object Identity

Each relation has an implicitly defined attribute named oid that contains the tuple's unique identifier, where each oid value is created and maintained by Postgres. The oid attribute can be accessed but not updated by user queries. Among other uses, the oid can be used as a mechanism to simulate attribute types that reference tuples in other relations. For example, we can define a type that references a tuple in the Staff relation as:

```
DEFINE TYPE Staff(int4) IS
  RETRIEVE (Staff.all) WHERE Staff.oid = $1
```

The relation name can be used for the type name because relations, types, and procedures have separate name spaces. An actual argument is supplied when a value is assigned to an attribute of type Staff. We can now create a relation that uses this reference type:

```
CREATE PropertyForRent(propertyNo = char[5], street = char[25], city = char[15],
        postcode = char[8], type = char[1], rooms = int2, rent = float4,
        ownerNo = char[5], branchNo = char[4], staffNo = Staff)
KEY(propertyNo)
```

The attribute staffNo represents the member of staff who oversees the rental of the property. The following query adds a property to the database:

```
APPEND PropertyForRent(propertyNo = "PA14", street = "16 Holhead",
        city = "Aberdeen", postcode = "AB7 5SU", type = "H", rooms = 6,
        rent = 650, ownerNo = "CO46", branchNo = "B007", staffNo = Staff(s.oid))
FROM s IN Staff
WHERE s.staffNo = "SA9")
```

28.4 SQL:1999 and SQL:2003

In Chapters 5 and 6 we provided an extensive tutorial on the features of the ISO SQL standard, concentrating mainly on those features present in the 1992 version of the standard, commonly referred to as SQL2 or SQL-92. ANSI (X3H2) and ISO (ISO/IEC JTC1/SC21/WG3) SQL standardization have added features to the SQL specification to support object-oriented data management, referred to as SQL:1999 (ISO, 1999a) and SQL:2003 (ISO, 2003a). As we mentioned earlier, the SQL:2003 standard is extremely large and comprehensive, and is divided into the following parts:

(1) *ISO/IEC 9075–1* SQL/Framework.

(2) *ISO/IEC 9075–2* SQL/Foundation, which includes new data types, user-defined types, rules and triggers, transactions, stored routines, and binding methods (embedded SQL, dynamic SQL, and direct invocation of SQL).

(3) *ISO/IEC 9075–3* SQL/CLI (Call-Level Interface), which specifies the provision of an API interface to the database, as we discuss in Appendix E, based on the SQL Access Group and X/Open's CLI definitions.

(4) *ISO/IEC 9075–4* SQL/PSM (Persistent Stored Modules), which allows procedures and user-defined functions to be written in a 3GL or in SQL and stored in the database, making SQL computationally complete.

(5) *ISO/IEC 9075–9* SQL/MED (Management of External Data), which defines extensions to SQL to support management of external data through the use of foreign tables and datalink data types.

(6) *ISO/IEC 9075–10* SQL/OLB (Object Language Bindings), which defines facilities for embedding SQL statements in Java programs.

(7) *ISO/IEC 9075–11* SQL/Schemata (Information and Definition Schemas), which defines two schemas INFORMATION_SCHEMA and DEFINITION_SCHEMA. The Information Schema defines views about database objects such as tables, views, and columns. These views are defined in terms of the base tables in the Definition Schema.

(8) *ISO/IEC 9075–13* SQL/JRT (Java Routines and Types Using the Java Programming Language), which defines extensions to SQL to allow the invocation of static methods written in Java as SQL-invoked routines, and to use classes defined in Java as SQL structured types.

(9) *ISO/IEC 9075–14* SQL/XML (XML-Related Specifications), which defines extensions to SQL to enable creation and manipulation of XML documents.

In this section we examine some of these features, covering:

- type constructors for row types and reference types;
- user-defined types (distinct types and structured types) that can participate in supertype/subtype relationships;
- user-defined procedures, functions, methods, and operators;
- type constructors for collection types (arrays, sets, lists, and multisets);
- support for large objects – Binary Large Objects (BLOBs) and Character Large Objects (CLOBs);
- recursion.

Many of the object-oriented concepts that we discussed in Section 25.3 are in the proposal. The definitive release of the SQL:1999 standard became significantly behind schedule and some of the features were deferred to a later version of the standard.

Row Types 28.4.1

A **row type** is a sequence of field name/data type pairs that provides a data type to represent the types of rows in tables, so that complete rows can be stored in variables, passed as arguments to routines, and returned as return values from function calls. A row type can also be used to allow a column of a table to contain row values. In essence, the row is a table nested within a table.

Example 28.1 Use of row type

To illustrate the use of row types, we create a simplified Branch table consisting of the branch number and address, and insert a record into the new table:

```
CREATE TABLE Branch (
        branchNo CHAR(4),
        address ROW(street      VARCHAR(25),
                    city        VARCHAR(15),
                    postcode    ROW(cityIdentifier  VARCHAR(4),
                                    subPart         VARCHAR(4))));
INSERT INTO Branch
VALUES ('B005', ROW('22 Deer Rd', 'London', ROW('SW1', '4EH')));
```

28.4.2 User-Defined Types

SQL:2003 allows the definition of **user-defined types** (UDTs), which we have previously referred to as abstract data types (ADTs). They may be used in the same way as the predefined types (for example, CHAR, INT, FLOAT). UDTs are subdivided into two categories: distinct types and structured types. The simpler type of UDT is the **distinct type**, which allows differentiation between the same underlying base types. For example, we could create the following two distinct types:

```
CREATE TYPE OwnerNumberType AS VARCHAR(5) FINAL;
CREATE TYPE StaffNumberType AS VARCHAR(5) FINAL;
```

If we now attempt to treat an instance of one type as an instance of the other type, an error would be generated. Note that although SQL also allows the creation of domains to distinguish between different data types, the purpose of an SQL domain is solely to constrain the set of valid values that can be stored in a column with that domain.

In its more general case, a UDT definition consists of one or more **attribute definitions**, zero or more **routine declarations** (methods) and, in a subsequent release, **operator declarations**. We refer to routines and operators generically as routines. In addition, we can also define the equality and ordering relationships for the UDT using the CREATE ORDERING FOR statement.

The value of an attribute can be accessed using the common dot notation (.). For example, assuming p is an instance of the UDT PersonType, which has an attribute fName of type VARCHAR, we can access the fName attribute as:

```
p.fName
p.fName = 'A. Smith'
```

Encapsulation and observer and mutator functions

SQL encapsulates each attribute of structured types by providing a pair of built-in routines that are invoked whenever a user attempts to reference the attribute, an **observer** (get) function and a **mutator** (set) function. The observer function returns the current value of the attribute; the mutator function sets the value of the attribute to a value specified as a parameter. These functions can be redefined by the user in the definition of the UDT. In this way, attribute values are encapsulated and are accessible to the user only by invoking these functions. For example, the observer function for the fName attribute of PersonType would be:

```
FUNCTION fName(p PersonType) RETURNS VARCHAR(15)
  RETURN p.fName;
```

and the corresponding mutator function to set the value to *newValue* would be:

```
FUNCTION fName(p PersonType RESULT, newValue VARCHAR(15))
  RETURNS PersonType
BEGIN
        p.fName = newValue;
        RETURN p;
END;
```

Constructor functions and the NEW expression

A (public) **constructor function** is automatically defined to create new instances of the type. The constructor function has the same name and type as the UDT, takes zero arguments, and returns a new instance of the type with the attributes set to their default value. User-defined **constructor methods** can be provided by the user to initialize a newly created instance of a structured type. Each method must have the same name as the structured type but the parameters must be different from the system-supplied constructor. In addition, each user-defined constructor method must differ in the number of parameters or in the data types of the parameters. For example, we could initialize a constructor for type PersonType as follows:

```
CREATE CONSTRUCTOR METHOD PersonType (fN VARCHAR(15),
  IN VARCHAR(15), sx CHAR) RETURNS PersonType
        BEGIN
          SET SELF.fName = fN;
          SET SELF.lName = lN;
          SET SELF.sex = sx;
          RETURN SELF;
        END;
```

The **NEW** expression can be used to invoke the system-supplied constructor function, for example:

```
SET p = NEW PersonType();
```

User-defined constructor methods must be invoked in the context of the NEW expression. For example, we can create a new instance of PersonType and invoke the above user-defined constructor method as follows:

```
SET p = NEW PersonType('John', 'White', 'M');
```

This is effectively translated into:

```
SET p = PersonType().PersonType('John', 'White', 'M');
```

Other UDT methods

Instances of UDTs can be constrained to exhibit specified ordering properties. The EQUALS ONLY BY and ORDER FULL BY clauses may be used to specify type-specific functions for comparing UDT instances. The ordering can be performed using methods that are qualified as:

- RELATIVE The relative method is a function that returns a 0 for equals, a negative value for less than, and a positive value for greater than.
- MAP The map method uses a function that takes a single argument of the UDT type and returns a predefined data type. Comparing two UDTs is achieved by comparing the two map values associated with them.
- STATE The state method compares the attributes of the operands to determine an order.

CAST functions can also be defined to provide user-specified conversion functions between different UDTs. In a subsequent version of the standard it may also be possible to override some of the built-in operators.

Example 28.2 Definition of a new UDT

To illustrate the creation of a new UDT, we create a UDT for a PersonType.

```
CREATE TYPE PersonType AS (
            dateOfBirth    DATE,
            fName          VARCHAR(15),
            lName          VARCHAR(15),
            sex            CHAR)
INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD age () RETURNS INTEGER,
INSTANCE METHOD age (DOB DATE) RETURNS PersonType;
CREATE INSTANCE METHOD age () RETURNS INTEGER
            FOR PersonType
            BEGIN
              RETURN /* age calculated from SELF.dateOfBirth */;
            END;
```

```
CREATE INSTANCE METHOD age (DOB DATE) RETURNS PersonType
        FOR PersonType
        BEGIN
          SELF.dateOfBirth = /* code to set dateOfBirth from DOB*/;
          RETURN SELF;
        END;
```

This example also illustrates the use of **stored** and **virtual attributes**. A **stored attribute** is the default type with an attribute name and data type. The data type can be any known data type, including other UDTs. In contrast, **virtual attributes** do not correspond to stored data, but to derived data. There is an implied virtual attribute age, which is derived using the (observer) age function and assigned using the (mutator) age function.[†] From the user's perspective, there is no distinguishable difference between a stored attribute and a virtual attribute – both are accessed using the corresponding observer and mutator functions. Only the designer of the UDT will know the difference.

The keyword INSTANTIABLE indicates that instances can be created for this type. If NOT INSTANTIABLE had been specified, we would not be able to create instances of this type, only from one of its subtypes. The keyword NOT FINAL indicates that we can create subtypes of this user-defined type. We discuss the clause REF IS SYSTEM GENERATED in Section 28.4.6.

Subtypes and Supertypes 28.4.3

SQL:2003 allows UDTs to participate in a subtype/supertype hierarchy using the UNDER clause. A type can have more than one subtype but currently only one supertype (that is, multiple inheritance is not supported). A subtype inherits all the attributes and behavior (methods) of its supertype and it can define additional attributes and methods like any other UDT and it can override inherited methods.

Example 28.3 Creation of a subtype using the UNDER clause

To create a subtype StaffType of the supertype PersonType we write:

```
CREATE TYPE StaffType UNDER PersonType AS (
        staffNo     VARCHAR(5),
        position    VARCHAR(10)     DEFAULT 'Assistant',
        salary      DECIMAL(7, 2),
        branchNo    CHAR(4))
        INSTANTIABLE
        NOT FINAL
        INSTANCE METHOD isManager () RETURNS BOOLEAN;
```

[†] Note that the function name age has been overloaded here. We discuss how SQL distinguishes between these two functions in Section 28.4.5.

```
CREATE INSTANCE METHOD isManager() RETURNS BOOLEAN
FOR StaffType
BEGIN
        IF SELF.position = 'Manager' THEN
            RETURN TRUE;
        ELSE
            RETURN FALSE;
        END IF
END)
```

StaffType as well as having the attributes defined within the CREATE TYPE, also includes the inherited attributes of PersonType, along with the associated observer and mutator functions and any specified methods. In particular, the clause REF IS SYSTEM GENERATED is also in effect inherited. In addition, we have defined an instance method isManager that checks whether the specified member of staff is a Manager. We show how this method can be used in Section 28.4.8.

An instance of a subtype is considered an instance of all its supertypes. SQL:2003 supports the concept of **substitutability**: that is, whenever an instance of a supertype is expected an instance of the subtype can be used in its place. The type of a UDT can be tested using the TYPE predicate. For example, given a UDT, Udt1 say, we can apply the following tests:

```
TYPE Udt1 IS OF (PersonType)          // Check Udt1 is the PersonType or any of its
                                         subtypes
TYPE Udt1 IS OF (ONLY PersonType)     // Check Udt1 is the PersonType
```

In SQL:2003, as in most programming languages, every instance of a UDT must be associated with exactly *one most specific type*, which corresponds to the lowest subtype assigned to the instance. Thus, if the UDT has more than one direct supertype, then there must be a single type to which the instance belongs, and that single type must be a subtype of all the types to which the instance belongs. In some cases, this can require the creation of a large number of types. For example, a type hierarchy might consist of a maximal supertype Person, with Student and Staff as subtypes; Student itself might have three direct subtypes: Undergraduate, Postgraduate, and PartTimeStudent, as illustrated in Figure 28.2(a). If an instance has the type Person and Student, then the most specific type in this case is Student, a non-leaf type, since Student is a subtype of Person. However, with the current type hierarchy an instance cannot have the type PartTimeStudent as well as staff, unless we create a type PTStudentStaff, as illustrated in Figure 28.2(b). The new leaf type, PTStudentStaff, is then the most specific type of this instance. Similarly, some of the full-time undergraduate and postgraduate students may work part time (as opposed to full-time employees being part-time students), and so we would also have to add subtypes for FTUGStaff and FTPGStaff. If we generalized this approach, we could potentially create a large number of subtypes. In some cases, a better approach may be to use inheritance at the level of tables as opposed to types, as we discuss shortly.

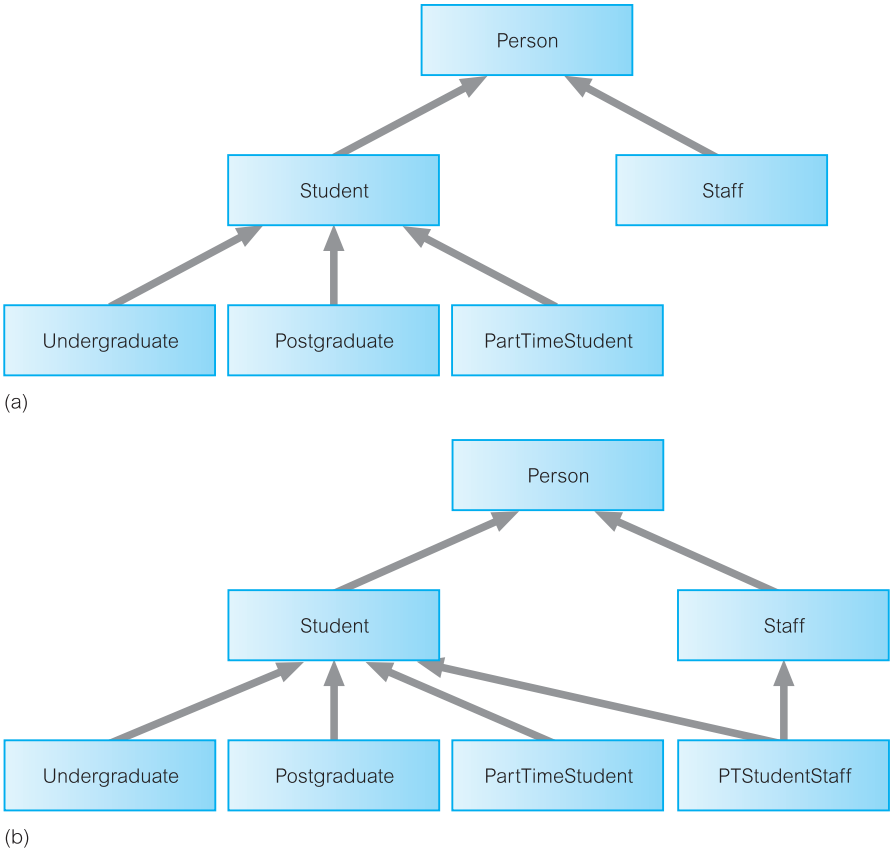


Figure 28.2
(a) Initial Student/Staff hierarchy; (b) modified Student/Staff hierarchy.

Privileges

To create a subtype, a user must have the UNDER privilege on the user-defined type specified as a supertype in the subtype definition. In addition, a user must have USAGE privilege on any user-defined type referenced within the new type.

Prior to SQL:1999, the SELECT privilege applied only to columns of tables and views. In SQL:1999, the SELECT privilege also applies to structured types, but only when instances of those types are stored in typed tables and only when the dereference operator is used from a REF value to the referenced row and then invokes a method on that referenced row. When invoking a method on a structured value that is stored in a column of any ordinary SQL table, SELECT privilege is required on that column. If the method is a mutator function, UPDATE privilege is also required on the column. In addition, EXECUTE privilege is required on all methods that are invoked.

User-Defined Routines 28.4.4

User-defined routines (UDRs) define methods for manipulating data and are an important adjunct to UDTs providing the required behavior for the UDTs. An ORDBMS should

provide significant flexibility in this area, such as allowing UDRs to return complex values that can be further manipulated (such as tables), and support for overloading of function names to simplify application development.

In SQL:2003, UDRs may be defined as part of a UDT or separately as part of a schema. An *SQL-invoked routine* may be a procedure, function, or method. It may be externally provided in a standard programming language such as 'C', C++, or Java, or defined completely in SQL using extensions that make the language computationally complete, as we discuss in Section 28.4.10.

An *SQL-invoked procedure* is invoked from an SQL CALL statement. It may have zero or more parameters, each of which may be an input parameter (IN), an output parameter (OUT), or both an input and output parameter (INOUT), and it has a body if it is defined fully within SQL. An *SQL-invoked function* returns a value; any specified parameters must be input parameters. One input parameter can be designated as the result (using the RESULT keyword), in which case the parameter's data type must match the type of the RETURNS type. Such a function is called *type-preserving*, because it always returns a value whose runtime type is the same as the most specific type (see Section 28.4.3) of the RETURN parameter (not some subtype of that type). Mutator functions are always type-preserving. An *SQL-invoked method* is similar to a function but has some important differences:

- a method is associated with a single UDT;
- the signature of every method associated with a UDT must be specified in that UDT and the definition of the method must specify that UDT (and must also appear in the same schema as the UDT).

There are three types of methods:

- *constructor methods*, which initialize a newly created instance of a UDT;
- *instance methods*, which operate on specific instances of a UDT;
- *static methods*, which are analogous to class methods in some object-oriented programming languages and operate at the UDT level rather than at the instance level.

In the first two cases, the methods include an additional implicit first parameter called SELF whose data type is that of the associated UDT. We saw an example of the SELF parameter in the user-defined constructor method for PersonType. A method can be invoked in one of three ways:

- a constructor method is invoked using the NEW expression, as discussed previously;
- an instance method is invoked using the standard dot notation, for example, p.fName, or using the generalized invocation format, for example, (p **AS** StaffType).fName();
- a static method is invoked using ::, for example, if totalStaff is a static method of StaffType, we could invoke it as StaffType::totalStaff().

An *external routine* is defined by specifying an external clause that identifies the corresponding 'compiled code' in the operating system's file storage. For example, we may wish to use a function that creates a thumbnail image for an object stored in the database. The functionality cannot be provided in SQL and so we have to use a function provided externally, using the following CREATE FUNCTION statement with an EXTERNAL clause:

```
CREATE FUNCTION thumbnail(IN myImage ImageType) RETURNS BOOLEAN
EXTERNAL NAME '/usr/dreamhome/bin/images/thumbnail'
LANGUAGE C
PARAMETER STYLE GENERAL
DETERMINISTIC
NO SQL;
```

This SQL statement associates the SQL function named thumbnail with an external file, 'thumbnail'. It is the user's responsibility to provide this compiled function. Thereafter, the ORDBMS will provide a method to dynamically link this object file into the database system so that it can be invoked when required. The procedure for achieving this is outside the bounds of the SQL standard and so is left as implementation-defined. A routine is *deterministic* if it always returns the same return value(s) for a given set of inputs. The NO SQL indicates that this function contains no SQL statements. The other options are READS SQL DATA, MODIFIES SQL DATA, and CONTAINS SQL.

Polymorphism 28.4.5

In Sections 25.3.7 and 25.3.8, we discussed the concepts of overriding, overloading, and more generally polymorphism. Different routines may have the same name, that is, routine names may be overloaded, for example to allow a UDT subtype to redefine a method inherited from a supertype, subject to the following constraints:

- No two functions in the same schema are allowed to have the same signature, that is, the same number of arguments, the same data types for each argument, and the same return type.
- No two procedures in the same schema are allowed to have the same name and the same number of parameters.

Overriding applies only to methods and then only based on the runtime value of the implicit SELF argument (note that a method definition has *parameters*, while a method invocation has *arguments*). SQL uses a generalized object model, so that the types of all arguments to a routine are taken into consideration when determining which routine to invoke, in order from left to right. Where there is not an exact match between the data type of an argument and the data type of the parameter specified, type precedence lists are used to determine the closest match. The exact rules for routine determination for a given invocation are quite complex and we do not give the full details here, but illustrate the mechanism for instance methods.

Instance method invocation

The mechanism for determining the appropriate invocation of an instance method is divided into two phases representing static analysis and runtime execution. In this section we provide an overview of these phases. The first phase proceeds as follows:

- All routines with the appropriate name are identified (all remaining routines are eliminated).

- All procedures/functions and all methods for which the user does not have EXECUTE privilege are eliminated.
- All methods that are not associated with the declared type (or subtype) of the implicit SELF argument are eliminated.
- All methods whose parameters are not equal to the number of arguments in the method invocation are eliminated.
- For the methods that remain, the system checks that the data type of each parameter matches the precedence list of the corresponding argument, eliminating those methods that do not match.
- If there are no candidate methods remaining a syntax error occurs.

For the remaining candidate methods the second (runtime) phase proceeds as follows:

- If the most specific type of the runtime value of the implicit argument to the method invocation has a type definition that includes one of the candidate methods, then that method is selected for execution.
- If the most specific type of the runtime value of the implicit argument to the method invocation has a type definition that does not include one of the candidate methods, then the method selected for execution is the candidate method whose associated type is the *nearest* supertype of all supertypes having such a method.

The argument values are converted to the parameter data types, if appropriate, and the body of the method is executed.

28.4.6 Reference Types and Object Identity

As discussed in Section 25.3.3 object identity is that aspect of an object which never changes and that distinguishes the object from all other objects. Ideally, an object's identity is independent of its name, structure, and location. The identity of an object persists even after the object has been deleted, so that it may never be confused with the identity of any other object. Other objects can use an object's identity as a unique way of referencing it.

Until SQL:1999, the only way to define relationships between tables was using the primary key/foreign key mechanism, which in SQL2 could be expressed using the referential table constraint clause REFERENCES, as discussed in Section 6.2.4. Since SQL:1999, **reference types** can be used to define relationships between row types and uniquely identify a row within a table. A reference type value can be stored in one table and used as a direct reference to a specific row in some base table that has been defined to be of this type (similar to the notion of a pointer type in 'C' or C++). In this respect, a reference type provides a similar functionality as the object identifier (OID) of object-oriented DBMSs, which we discussed in Section 25.3.3. Thus, references allow a row to be shared among multiple tables and enable users to replace complex join definitions in queries with much simpler path expressions. References also give the optimizer an alternative way to navigate data instead of using value-based joins.

REF IS SYSTEM GENERATED in a CREATE TYPE statement indicates that the actual values of the associated REF type are provided by the system, as in the PersonType created in Example 28.2. Other options are available but we omit the details here; the default is REF IS SYSTEM GENERATED. As we see shortly, a base table can be created to be of some structured type. Other columns can be specified for the table but at least one column must be specified, namely a column of the associated REF type, using the clause REF IS <columnName> SYSTEM GENERATED. This column is used to contain unique identifiers for the rows of the associated base table. The identifier for a given row is assigned when the row is inserted into the table and remains associated with that row until it is deleted.

Creating Tables 28.4.7

To maintain upwards compatibility with the SQL2 standard, it is still necessary to use the CREATE TABLE statement to create a table, even if the table consists of a single UDT. In other words, a UDT instance can persist only if it is stored as the column value in a table. There are several variations of the CREATE TABLE statement, as Examples 28.4–28.6 illustrate.

Example 28.4 Creation of a table based on a UDT

To create a table using the PersonType UDT, we could write:

```
CREATE TABLE Person (
        info PersonType
        CONSTRAINT DOB_Check CHECK(dateOfBirth > DATE '1900-01-01'));
```

or

```
CREATE TABLE Person OF PersonType (
        dateOfBirth WITH OPTIONS
        CONSTRAINT DOB_Check CHECK (dateOfBirth > DATE '1900-01-01')
        REF IS PersonID SYSTEM GENERATED);
```

In the first instance, we would access the columns of the Person table using a path expression such as 'Person.info.fName'; in the second version, we would access the columns using a path expression such as 'Person.fName'.

Example 28.5 Using a reference type to define a relationship

In this example, we model the relationship between PropertyForRent and Staff using a reference type.

```
CREATE TABLE PropertyForRent(
    propertyNo    PropertyNumber    NOT NULL,
    street        Street            NOT NULL,
    city          City              NOT NULL,
    postcode      PostCode,
    type          PropertyType      NOT NULL   DEFAULT 'F',
    rooms         PropertyRooms     NOT NULL   DEFAULT 4,
    rent          PropertyRent      NOT NULL   DEFAULT 600,
    staffID       REF(StaffType)    SCOPE Staff
                  REFERENCES ARE CHECKED ON DELETE CASCADE,
    PRIMARY KEY (propertyNo));
```

In Example 6.1 we modeled the relationship between PropertyForRent and Staff using the traditional primary key/foreign key mechanism. Here, however, we have used a reference type, REF(StaffType), to model the relationship. The SCOPE clause specifies the associated referenced table. REFERENCES ARE CHECKED indicates that referential integrity is to be maintained (alternative is REFERENCES ARE NOT CHECKED). ON DELETE CASCADE corresponds to the normal referential action that existed in SQL2. Note that an ON UPDATE clause is not required, as the column staffID in the Staff table cannot be updated.

SQL:2003 does not provide a mechanism to store all instances of a given UDT unless the user explicitly creates a single table in which all instances are stored. Thus, in SQL:2003 it may not be possible to apply an SQL query to all instances of a given UDT. For example, if we created a second table such as:

```
CREATE TABLE Client (
    info          PersonType,
    prefType      CHAR,
    maxRent       DECIMAL(6, 2),
    branchNo      VARCHAR(4) NOT NULL);
```

then the instances of PersonType are now distributed over two tables: Staff and Client. This problem can be overcome in this particular case using the table inheritance mechanism, which allows a table to be created that inherits all the columns of an existing table using the UNDER clause. As would be expected, a subtable inherits every column from its supertable. Note that all the tables in a table hierarchy must have corresponding types that are in the same type hierarchy, and the tables in the table hierarchy must be in the same relative positions as the corresponding types in the type hierarchy. However, not every type in the type hierarchy has to be represented in the table hierarchy, provided the range of types for which tables are defined is contiguous. For example, referring to Figure 28.2(a), it would be legal to create tables for all types except Staff; however, it would be illegal to create tables for Person and Postgraduate without creating one for Student. Note also that additional columns cannot be defined as part of the subtable definition.

Example 28.6 Creation of a subtable using the UNDER clause

We can create a table for staff using table inheritance:

```
CREATE TABLE Staff OF StaffType UNDER Person;
```

When we insert rows into the Staff table, the values of the inherited columns are inserted into the Person table. Similarly, when we delete rows from the Staff table, the rows disappear from both the Staff and Person tables. As a result, when we access all rows of Person, this will also include all Staff details.

There are restrictions on the population of a table hierarchy:

- Each row of the supertable Person can correspond to at most one row in Staff.
- Each row in Staff must have exactly one corresponding row in Person.

The semantics maintained are those of *containment*: a row in a subtable is in effect 'contained' in its supertables. We would expect the SQL INSERT, UPDATE, and DELETE statements to maintain this consistency when the rows of subtables and supertables are being modified, as follows (at least conceptually):

- When a row is inserted into a subtable, then the values of any inherited columns of the table are inserted into the corresponding supertables, cascading upwards in the table hierarchy. For example, referring back to Figure 28.2(b), if we insert a row into PTStudentStaff, then the values of the inherited columns are inserted into Student and Staff, and then the values of the inherited columns of Student/Staff are inserted into Person.
- When a row is updated in a subtable, a similar procedure to the above is carried out to update the values of inherited columns in the supertables.
- When a row is updated in a supertable, then the values of all inherited columns in all corresponding rows of its direct and indirect subtables are also updated accordingly. As the supertable may itself be a subtable, the previous condition will also have to be applied to ensure consistency.
- When a row is deleted in a subtable/supertable, the corresponding rows in the table hierarchy are deleted. For example, if we deleted a row of Student, the corresponding rows of Person and Undergraduate/Postgraduate/PartTimeStudent/PTStudentStaff are deleted.

Privileges

As with the privileges required to create a new subtype, a user must have the UNDER privilege on the referenced supertable. In addition, a user must have USAGE privilege on any user-defined type referenced within the new table.

28.4.8 Querying Data

SQL:2003 provides the same syntax as SQL2 for querying and updating tables, with various extensions to handle objects. In this section, we illustrate some of these extensions.

Example 28.7 Retrieve a specific column, specific rows

Find the names of all Managers.

```
SELECT s.lName
FROM Staff s
WHERE s.position = 'Manager';
```

This query invokes the implicitly defined observer function position in the WHERE clause to access the position column.

Example 28.8 Invoking a user-defined function

Find the names and ages of all Managers.

```
SELECT s.lName, s.age
FROM Staff s
WHERE s.isManager;
```

This alternative method of finding Managers uses the user-defined method isManager as a predicate of the WHERE clause. This method returns the boolean value TRUE if the member of staff is a manager (see Example 28.3). In addition, this query also invokes the inherited virtual (observer) function age as an element of the SELECT list.

Example 28.9 Use of ONLY to restrict selection

Find the names of all people in the database over 65 years of age.

```
SELECT p.lName, p.fName
FROM Person p
WHERE p.age > 65;
```

This query lists not only the details of rows that have been explicitly inserted into the Person table, but also the names from any rows that have been inserted into any direct or indirect subtables of Person, in this case, Staff and Client.

Suppose, however, that rather than wanting the details of all people, we want only the details of the specific instances of the Person table, excluding any subtables. This can be achieved using the ONLY keyword:

```
SELECT p.lName, p.fName
FROM ONLY (Person) p
WHERE p.age > 65;
```

Example 28.10 Use of the dereference operator

Find the name of the member of staff who manages property 'PG4'.

```
SELECT p.staffID–>fName AS fName, p.staffID–>lName AS lName
FROM PropertyForRent p
WHERE p.propertyNo = 'PG4';
```

References can be used in path expressions that permit traversal of object references to navigate from one row to another. To traverse a reference, the dereference operator (–>) is used. In the SELECT statement, p.staffID is the normal way to access a column of a table. In this particular case though, the column is a reference to a row of the Staff table, and so we must use the dereference operator to access the columns of the dereferenced table. In SQL2, this query would have required a join or nested subquery.

To retrieve the member of staff for property PG4, rather than just the first and last names, we would use the following query instead:

```
SELECT DEREF(p.staffID) AS Staff
FROM PropertyForRent p
WHERE p.propertyNo = 'PG4';
```

Although reference types are similar to foreign keys, there are significant differences. In SQL:2003, referential integrity is maintained only by using a referential constraint definition specified as part of the table definition. By themselves, reference types do not provide referential integrity. Thus, the SQL reference type should not be confused with that provided in the ODMG object model. In the ODMG model, OIDs are used to model relationships between types and referential integrity is automatically defined, as discussed in Section 27.2.2.

Collection Types 28.4.9

Collections are type constructors that are used to define collections of other types. Collections are used to store multiple values in a single column of a table and can result in nested tables where a column in one table actually contains another table. The result can be a single table that represents multiple master-detail levels. Thus, collections add flexibility to the design of the physical database structure.

SQL:1999 introduced an ARRAY collection type and SQL:2003 added the MULTISET collection type, and a subsequent version of the standard may introduce parameterized LIST and SET collection types. In each case, the parameter, called the *element type*, may be a predefined type, a UDT, a row type, or another collection, but cannot be a reference type or a UDT containing a reference type. In addition, each collection must be homogeneous: all elements must be of the same type, or at least from the same type hierarchy. The collection types have the following meaning:

- ARRAY – one-dimensional array with a maximum number of elements;
- MULTISET – unordered collection that does allow duplicates;
- LIST – ordered collection that allows duplicates;
- SET – unordered collection that does not allow duplicates.

These types are similar to those defined in the ODMG 3.0 standard discussed in Section 27.2, with the name Bag replaced with the SQL MULTISET.

ARRAY collection type

An array is an ordered collection of not necessarily distinct values, whose elements are referenced by their ordinal position in the array. An array is declared by a data type and, optionally, a maximum cardinality; for example:

VARCHAR(25) **ARRAY**[5]

The elements of this array can be accessed by an index ranging from 1 to the maximum cardinality (the function CARDINALITY returns the number of current elements in the array). Two arrays of comparable types are considered identical if and only if they have the same cardinality and every ordinal pair of elements is identical.

An array type is specified by an array type constructor, which can be defined by enumerating the elements as a comma-separated list enclosed in square brackets or by using a query expression with degree 1; for example:

ARRAY [‘Mary White’, ‘Peter Beech’, ‘Anne Ford’, ‘John Howe’, ‘Alan Brand’]
ARRAY (SELECT rooms **FROM** PropertyForRent**)**

In these cases, the data type of the array is determined by the data types of the various array elements.

Example 28.11 Use of a collection ARRAY

To model the requirement that a branch has up to three telephone numbers, we could implement the column as an ARRAY collection type:

telNo **VARCHAR**(13) **ARRAY**[3]

We could now retrieve the first telephone number at branch B003 using the following query:

```
SELECT telNo[1]
FROM Branch
WHERE branchNo = 'B003';
```

MULTISET collection type

A multiset is an unordered collection of elements, all of the same type, with duplicates permitted. Since a multiset is unordered there is no ordinal position to reference individual elements of a multiset. Unlike arrays, a multiset is an unbounded collection with no declared maximum cardinality (although there will be an implementation-defined limit). Although multisets are analogous to tables, they are not regarded as the same as tables, and operators are provided to convert a multiset to a table (UNNEST) and a table to a multiset (MULTISET).

There is no separate type proposed for sets at present. Instead, a set is simply a special kind of multiset, namely one that has no duplicate elements. A predicate is provided to check whether a multiset is a set.

Two multisets of comparable element types, A and B say, are considered identical if and only if they have the same cardinality and for each element x in A, the number of elements of A that are identical to x, including x itself, equals the number of elements of B that are equal to x. Again as with array types, a multiset type constructor can be defined by enumerating their elements as a comma-separated list enclosed in square brackets, or by using a query expression with degree 1, or by using a table value constructor.

Operations on multisets include:

- The SET function to remove duplicates from a multiset to produce a set.
- The CARDINALITY function to return the number of current elements.
- The ELEMENT function to return the element of a multiset if the multiset only has one element (or null if the multiset has no elements). An exception is raised if the multiset has more than one element.
- MULTISET UNION, which computes the union of two multisets; the keywords ALL or DISTINCT can be specified to either retain duplicates or remove them.
- MULTISET INTERSECT, which computes the intersection of two multisets; the keyword DISTINCT can be specified to remove duplicates; the keyword ALL can be specified to place in the result as many instances of each value as the minimum number of instances of that value in either operand.
- MULTISET EXCEPT, which computes the difference of two multisets; again, the keyword DISTINCT can be specified to remove duplicates; the keyword ALL can be specified to place in the result a number of instances of a value, equal to the number of instances of the value in the first operand minus the number of instances of the second operand.

There are three new aggregate functions for multisets:

- COLLECT, which creates a multiset from the value of the argument in each row of a group;

- FUSION, which creates a multiset union of a multiset value in all rows of a group;
- INTERSECTION, which creates the multiset intersection of a multiset value in all rows of a group.

In addition, a number of predicates exist for use with multisets:

- comparison predicate (equality and inequality only);
- DISTINCT predicate;
- MEMBER predicate;
- SUBMULTISET predicate, which tests whether one multiset is a submultiset of another;
- IS A SET/IS NOT A SET predicate, which checks whether a multiset is a set.

Example 28.12 Use of a collection MULTISET

Extend the Staff *table to contain the details of a number of next-of-kin and then find the first and last names of John White's next-of-kin.*

We include the definition of a nextOfKin column in Staff as follows (NameType contains a fName and lName attribute):

```
nextOfKin    NameType MULTISET
```

The query becomes:

```
SELECT n.fName, n.lName
FROM Staff s, UNNEST (s.nextOfKin) AS n(fName, lName)
WHERE s.lName = 'White' AND s.fName = 'John';
```

Note that in the FROM clause we may use the multiset-valued field s.nextOfKin as a table reference.

Example 28.13 Use of the FUSION and INTERSECTION aggregate functions

Consider the following table, PropertyViewDates, giving the dates properties have been viewed by potential renters:

propertyNo	viewDates
PA14	MULTISET['14-May-04', '24-May-04']
PG4	MULTISET['20-Apr-04', '14-May-04', '26-May-04']
PG36	MULTISET['28-Apr-04', '14-May-04']
PL94	Null

The following query based on multiset aggregation:

```
SELECT FUSION(viewDates) AS viewDateFusion,
       INTERSECTION(viewDates) AS viewDateIntersection
FROM PropertyViewDates;
```

produces the following result set:

viewDateFusion	viewDateIntersection
MULTISET['14-May-04', '14-May-04', '14-May-04', '24-May-04', '20-Apr-04', '26-May-04', '28-Apr-04']	MULTISET['14-May-04']

The fusion is computed by first discarding those rows with a null (in this case, the row for property PL94). Then each member of each of the remaining three multisets is copied to the result set. The intersection is computed by again discarding those rows with a null and then finding the duplicates in the input multisets.

Typed Views 28.4.10

SQL:2003 also supports **typed views**, sometimes called *object views* or *referenceable views*. A typed view is created based on a particular structured type and a subview can be created based on this typed view. The following example illustrates the usage of typed views.

Example 28.14 Creation of typed views

The following statements create two views based on the PersonType and StaffType structured types.

```
CREATE VIEW FemaleView OF PersonType (REF IS personID DERIVED)
    AS SELECT fName, lName
    FROM ONLY (Person)
    WHERE sex = 'F';

CREATE VIEW FemaleStaff3View OF StaffType UNDER FemaleView
    AS SELECT fName, lName, staffNo, position
    FROM ONLY (Staff)
    WHERE branchNo = 'B003';
```

The (REF IS personID DERIVED) is the self-referencing column specification discussed previously. When defining a subview this clause cannot be specified. When defining a

maximal superview this clause can be specified, although the option SYSTEM GENERATED cannot be used, only USER GENERATED or DERIVED. If USER GENERATED is specified, then the degree of the view is one more than the number of attributes of the associated structured type; if DERIVED is specified then the degree is the same as the number of attributes in the associated structured type and no additional self-referencing column is included.

As with normal views, new column names can be specified as can the WITH CHECK OPTION clause.

28.4.11 Persistent Stored Modules

A number of new statement types have been added to SQL to make the language computationally complete, so that object behavior (methods) can be stored and executed from within the database as SQL statements (ISO, 1999b; 2003b). Statements can be grouped together into a compound statement (block), with its own local variables. Some of the additional statements provided are:

- An assignment statement that allows the result of an SQL value expression to be assigned to a local variable, a column, or an attribute of a UDT. For example:

```
DECLARE b BOOLEAN;
DECLARE staffMember StaffType;
b = staffMember.isManager;
```

- An IF . . . THEN . . . ELSE . . . END IF statement that allows conditional processing. We saw an example of this statement in the isManager method of Example 28.3.
- A CASE statement that allows the selection of an execution path based on a set of alternatives. For example:

```
CASE lowercase(x)
        WHEN 'a'        THEN SET x = 1;
        WHEN 'b'        THEN SET x = 2;
                             SET y = 0;
        WHEN 'default'  THEN SET x = 3;
END CASE;
```

- A set of statements that allows repeated execution of a block of SQL statements. The iterative statements are FOR, WHILE, and REPEAT, examples of which are:

```
FOR x, y AS SELECT a, b FROM Table1 WHERE searchCondition DO
   . . .
END FOR;
```

```
WHILE b <> TRUE DO
   . . .
END WHILE;
```

```
REPEAT
   . . .
UNTIL b <> TRUE
END REPEAT;
```

- A CALL statement that allows procedures to be invoked and a RETURN statement that allows an SQL value expression to be used as the return value from an SQL function or method.

Condition handling

The SQL Persistent Stored Module (SQL/PSM) language includes condition handling to handle exceptions and completion conditions. Condition handling works by first defining a handler by specifying its type, the exception and completion conditions it can resolve, and the action it takes to do so (an SQL procedure statement). Condition handling also provides the ability to explicitly signal exception and completion conditions, using the SIGNAL/RESIGNAL statement.

A handler for an associated exception or completion condition can be declared using the DECLARE . . . HANDLER statement:

```
DECLARE {CONTINUE | EXIT | UNDO} HANDLER
FOR SQLSTATE {sqlstateValue | conditionName | SQLEXCEPTION |
              SQLWARNING | NOT FOUND} handlerAction;
```

A condition name and an optional corresponding SQLSTATE value can be declared using:

```
DECLARE conditionName CONDITION
[FOR SQLSTATE sqlstateValue]
```

and an exception condition can be signaled or resignaled using:

```
SIGNAL sqlstateValue; or RESIGNAL sqlstateValue;
```

When a compound statement containing a handler declaration is executed, a handler is created for the associated conditions. A handler is *activated* when it is the most appropriate handler for the condition that has been raised by the SQL statement. If the handler has specified CONTINUE, then on activation it will execute the handler action before returning control to the compound statement. If the handler type is EXIT, then after executing the handler action, the handler leaves the compound statement. If the handler type is UNDO, then the handler rolls back all changes made within the compound statement, executes the associated handler action, and then returns control to the compound statement. If the handler does not complete with a *successful completion* condition, then an implicit resignal is executed, which determines whether there is another handler that can resolve the condition.

Triggers 28.4.12

As we discussed in Section 8.2.7, a **trigger** is an SQL (compound) statement that is executed automatically by the DBMS as a side effect of a modification to a named table. It is

similar to an SQL routine, in that it is a named SQL block with declarative, executable, and condition-handling sections. However, unlike a routine, a trigger is executed implicitly whenever the *triggering event* occurs, and a trigger does not have any arguments. The act of executing a trigger is sometimes known as *firing* the trigger. Triggers can be used for a number of purposes including:

- validating input data and maintaining complex integrity constraints that otherwise would be difficult, if not impossible, through table constraints;
- supporting alerts (for example, using electronic mail) that action needs to be taken when a table is updated in some way;
- maintaining audit information, by recording the changes made, and by whom;
- supporting replication, as discussed in Chapter 24.

The basic format of the CREATE TRIGGER statement is as follows:

```
CREATE TRIGGER TriggerName
        BEFORE | AFTER <triggerEvent> ON <TableName>
        [REFERENCING <oldOrNewValuesAliasList>]
        [FOR EACH {ROW | STATEMENT}]
        [WHEN (triggerCondition)]
        <triggerBody>
```

Triggering events include insertion, deletion, and update of rows in a table. In the latter case only, a triggering event can also be set to cover specific named columns of a table. A trigger has an associated timing of either BEFORE or AFTER. A BEFORE trigger is fired before the associated event occurs and an AFTER trigger is fired after the associated event occurs. The triggered action is an SQL procedure statement, which can be executed in one of two ways:

- for each row affected by the event (FOR EACH ROW). This is called a row-level trigger;
- only once for the entire event (FOR EACH STATEMENT), which is the default. This is called a statement-level trigger.

The <oldOrNewValuesAliasList> can refer to:

- an old or new row (OLD/NEW or OLD ROW/NEW ROW), in the case of a row-level trigger;
- an old or new table (OLD TABLE/NEW TABLE), in the case of an AFTER trigger.

Clearly, old values are not applicable for insert events, and new values are not applicable for delete events. The body of a trigger cannot contain any:

- SQL transaction statements, such as COMMIT or ROLLBACK;
- SQL connection statements, such as CONNECT or DISCONNECT;
- SQL schema definition or manipulation statements, such as the creation or deletion of tables, user-defined types, or other triggers;
- SQL session statements, such as SET SESSION CHARACTERISTICS, SET ROLE, SET TIME ZONE.

Furthermore, SQL does not allow *mutating triggers*, that is, triggers that cause a change resulting in the same trigger to be invoked again, possibly in an endless loop. As more than one trigger can be defined on a table, the order of firing of triggers is important. Triggers are fired as the trigger event (INSERT, UPDATE, DELETE) is executed. The following order is observed:

(1) Execution of any BEFORE statement-level trigger on the table.

(2) For each row affected by the statement:

  (a) execution of any BEFORE row-level trigger;
  (b) execution of the statement itself;
  (c) application of any referential constraints;
  (d) execution of any AFTER row-level trigger.

(3) Execution of any AFTER statement-level trigger on the table.

Note from this ordering that BEFORE triggers are activated before referential integrity constraints have been checked. Thus, it is possible that the requested change that has caused the trigger to be invoked will violate database integrity constraints and will have to be disallowed. Therefore, BEFORE triggers should not further modify the database.

Should there be more than one trigger on a table with the same trigger event and the same action time (BEFORE or AFTER) then the SQL standard specifies that the triggers are executed in the order they were created. We now illustrate the creation of triggers with some examples.

Example 28.15 Use of an AFTER INSERT trigger

Create a set of mailshot records for each new PropertyForRent *row. For the purposes of this example, assume that there is a* Mailshot *table that records prospective renter details and property details.*

```
CREATE TRIGGER InsertMailshotTable
    AFTER INSERT ON PropertyForRent
    REFERENCING NEW ROW AS pfr
    BEGIN ATOMIC
        INSERT INTO Mailshot VALUES
            (SELECT c.fName, c.lName, c.maxRent, pfr.propertyNo, pfr.street,
                    pfr.city, pfr.postcode, pfr.type, pfr.rooms, pfr.rent
             FROM Client c
             WHERE c.branchNo = pfr.branchNo AND
                    (c.prefType = pfr.type AND c.maxRent <= pfr.rent))
    END;
```

This trigger is executed after the new row has been inserted. The FOR EACH clause has been omitted, defaulting to FOR EACH STATEMENT, as an INSERT statement only inserts one row at a time. The body of the trigger is an INSERT statement based on a subquery that finds all matching client rows.

Example 28.16 Use of an AFTER INSERT trigger with condition

Create a trigger that modifies all current mailshot records if the rent for a property changes.

```
CREATE TRIGGER UpdateMailshotTable
    AFTER UPDATE OF rent ON PropertyForRent
    REFERENCING NEW ROW AS pfr
    FOR EACH ROW
    BEGIN ATOMIC
        DELETE FROM Mailshot WHERE maxRent > pfr.rent;
        UPDATE Mailshot SET rent = pfr.rent
        WHERE propertyNo = pfr.propertyNo;
    END;
```

This trigger is executed after the rent field of a PropertyForRent row has been updated. The FOR EACH ROW clause is specified, as all property rents may have been increased in one UPDATE statement, for example due to a cost of living rise. The body of the trigger has two SQL statements: a DELETE statement to delete those mailshot records where the new rental price is outside the client's price range, and an UPDATE statement to record the new rental price in all rows relating to that property.

Triggers can be a very powerful mechanism if used appropriately. The major advantage is that standard functions can be stored within the database and enforced consistently with each update to the database. This can dramatically reduce the complexity of applications. However, there can be some disadvantages:

- *Complexity* When functionality is moved from the application to the database, the database design, implementation, and administration tasks become more complex.
- *Hidden functionality* Moving functionality to the database and storing it as one or more triggers can have the effect of hiding functionality from the user. While this can simplify things for the user, unfortunately it can also have side effects that may be unplanned, and potentially unwanted and erroneous. The user no longer has control over what happens to the database.
- *Performance overhead* When the DBMS is about to execute a statement that modifies the database, it now has to evaluate the trigger condition to check whether a trigger should be fired by the statement. This has a performance implication on the DBMS. Clearly, as the number of triggers increases, this overhead also increases. At peak times, this overhead may create performance problems.

Privileges

To create a trigger, a user must have TRIGGER privilege on the specified table, SELECT privilege on any tables referenced in the triggerCondition of the WHEN clause, together with any privileges required to execute the SQL statements in the trigger body.

Large Objects 28.4.13

A **large object** is a data type that holds a large amount of data, such as a long text file or a graphics file. There are three different types of large object data types defined in SQL:2003:

- Binary Large Object (BLOB), a binary string that does not have a character set or collation association;
- Character Large Object (CLOB) and National Character Large Object (NCLOB), both character strings.

The SQL large object is slightly different from the original type of BLOB that appears in some database systems. In such systems, the BLOB is a non-interpreted byte stream, and the DBMS does not have any knowledge concerning the content of the BLOB or its internal structure. This prevents the DBMS from performing queries and operations on inherently rich and structured data types, such as images, video, word processing documents, or Web pages. Generally, this requires that the entire BLOB be transferred across the network from the DBMS server to the client before any processing can be performed. In contrast, the SQL large object does allow some operations to be carried out in the DBMS server. The standard string operators, which operate on characters strings and return character strings, also operate on character large object strings, such as:

- The concatenation operator, (string1 || string2), which returns the character string formed by joining the character string operands in the specified order.
- The character substring function, SUBSTRING(string FROM startpos FOR length), which returns a string extracted from a specified string from a start position for a given length.
- The character overlay function, OVERLAY(string1 PLACING string2 FROM startpos FOR length), which replaces a substring of string1, specified as a starting position and a length, with string2. This is equivalent to: SUBSTRING(string1 FROM 1 FOR length − 1) || string2 || SUBSTRING (string1 FROM startpos + length).
- The fold functions, UPPER(string) and LOWER(string), which convert all characters in a string to upper/lower case.
- The trim function, TRIM([LEADING | TRAILING | BOTH string1 FROM] string2), which returns string2 with leading and/or trailing string1 characters removed. If the FROM clause is not specified, all leading and training spaces are removed from string2.
- The length function, CHAR_LENGTH(string), which returns the length of the specified string.
- The position function, POSITION(string1 IN string2), which returns the start position of string1 within string2.

However, CLOB strings are not allowed to participate in most comparison operations, although they can participate in a LIKE predicate, and a comparison or quantified comparison predicate that uses the equals (=) or not equals (<>) operators. As a result of these restrictions, a column that has been defined as a CLOB string cannot be referenced in such

places as a GROUP BY clause, an ORDER BY clause, a unique or referential constraint definition, a join column, or in one of the set operations (UNION, INTERSECT, and EXCEPT).

A binary large object (BLOB) string is defined as a sequence of octets. All BLOB strings are comparable by comparing octets with the same ordinal position. The following operators operate on BLOB strings and return BLOB strings, and have similar functionality as those defined above:

- the BLOB concatenation operator (||);
- the BLOB substring function (SUBSTRING);
- the BLOB overlay function (OVERLAY);
- the BLOB trim function (TRIM).

In addition, the BLOB_LENGTH and POSITION functions and the LIKE predicate can also be used with BLOB strings.

Example 28.17 Use of Character and Binary Large Objects

Extend the Staff *table to hold a resumé and picture for the staff member.*

```
ALTER TABLE Staff
    ADD COLUMN resume CLOB(50K);
ALTER TABLE Staff
    ADD COLUMN picture BLOB(12M);
```

Two new columns have been added to the Staff table: resume, which has been defined as a CLOB of length 50K, and picture, which has been defined as a BLOB of length 12M. The length of a large object is given as a numeric value with an optional specification of K, M, or G, indicating kilobytes, megabytes, or gigabytes, respectively. The default length, if left unspecified, is implementation-defined.

28.4.14 Recursion

In Section 25.2 we discussed the difficulty that RDBMSs have with handling recursive queries. A major new operation in SQL for specifying such queries is **linear recursion**. To illustrate the new operation, we use the example given in Section 25.2 with the simplified Staff relation shown in Figure 25.1(a), which stores staff numbers and the corresponding manager's staff number. To find all the managers of all staff, we can use the following recursive query in SQL:2003:

```
WITH RECURSIVE
AllManagers (staffNo, managerStaffNo) AS
   (SELECT staffNo, managerStaffNo
    FROM Staff
```

```
    UNION
    SELECT in.staffNo, out.managerStaffNo
    FROM AllManagers in, Staff out
    WHERE in.managerStaffNo = out.staffNo);
SELECT * FROM AllManagers
ORDER BY staffNo, managerStaffNo;
```

This query creates a result table AllManagers with two columns staffNo and managerStaffNo containing all the managers of all staff. The UNION operation is performed by taking the union of all rows produced by the inner block until no new rows are generated. Note, if we had specified UNION ALL, any duplicate values would remain in the result table.

In some situations, an application may require the data to be inserted into the result table in a certain order. The recursion statement allows the specification of two orderings:

- depth-first, where each 'parent' or 'containing' item appears in the result before the items that it contains, as well as before its 'siblings' (items with the same parent or container);
- breadth-first, where items follow their 'siblings' without following the siblings' children.

For example, at the end of the WITH RECURSIVE statement we could add the following clause:

```
SEARCH BREADTH FIRST BY staffNo, managerStaffNo
    SET orderColumn
```

The SET clause identifies a new column name (orderColumn), which is used by SQL to order the result into the required breadth-first traversal.

If the data can be recursive, not just the data structure, an infinite loop can occur unless the cycle can be detected. The recursive statement has a CYCLE clause that instructs SQL to record a specified value to indicate that a new row has already been added to the result table. Whenever a new row is found, SQL checks that the row has not been added previously by determining whether the row has been marked with the specified value. If it has, then SQL assumes a cycle has been encountered and stops searching for further result rows. An example of the CYCLE clause is:

```
CYCLE staffNo, managerStaffNo
    SET cycleMark TO 'Y' DEFAULT 'N'
    USING cyclePath
```

cycleMark and cyclePath are user-defined column names for SQL to use internally. cyclePath is an ARRAY with cardinality sufficiently large to accommodate the number of rows in the result and whose element type is a row type with a column for each column in the cycle column list (staffNo and managerStaffNo in our example). Rows satisfying the query are cached in cyclePath. When a row satisfying the query is found for the first time (which can be determined by its absence from cyclePath), the value of the cycleMark column is set to 'N'. When the same row is found again (which can be determined by its presence in cyclePath), the cycleMark column of the existing row in the result table is modified to the cycleMark value of 'Y' to indicate that the row starts a cycle.

28.5 Query Processing and Optimization

In the previous section we introduced some features of the new SQL standard, although some of the features, such as collections, has been deferred to a later version of the standard. These features address many of the weaknesses of the relational model that we discussed in Section 25.2. Unfortunately, the SQL:2003 standard does not address some areas of extensibility, so implementation of features such as the mechanism for defining new index structures and giving the query optimizer cost information about user-defined functions will vary among products. The lack of a standard way for third-party vendors to integrate their software with multiple ORDBMSs demonstrates the need for standards beyond the focus of SQL:2003. In this section we explore why these mechanisms are important for a true ORDBMS using a series of illustrative examples.

Example 28.18 Use of user-defined functions revisited

List the flats that are for rent at branch B003.

We might decide to implement this query using a function, defined as follows:

```
CREATE FUNCTION flatTypes() RETURNS SET(PropertyForRent)
    SELECT * FROM PropertyForRent WHERE type = 'Flat';
```

and the query becomes:

```
SELECT propertyNo, street, city, postcode
FROM TABLE (flatTypes())
WHERE branchNo = 'B003';
```

In this case, we would hope that the query processor would be able to 'flatten' this query using the following steps:

(1)
```
SELECT propertyNo, street, city, postcode
FROM TABLE (SELECT * FROM PropertyForRent WHERE type = 'Flat')
WHERE branchNo = 'B003';
```

(2)
```
SELECT propertyNo, street, city, postcode
FROM PropertyForRent
WHERE type = 'Flat' AND branchNo = 'B003';
```

If the PropertyForRent table had a B-tree index on the branchNo column, for example, then the query processor should be able to use an indexed scan over branchNo to efficiently retrieve the appropriate rows, as discussed in Section 21.4.

From this example, one capability we require is that the ORDBMS query processor flattens queries whenever possible. This was possible in this case because our user-defined function had been implemented in SQL. However, suppose that the function had been defined as an external function. How would the query processor know how to optimize this query? The answer to this question lies in an extensible query optimization mechanism. This may

require the user to provide a number of routines specifically for use by the query optimizer in the definition of a new ADT. For example, the Illustra ORDBMS, now part of Informix, requires the following information when an (external) user-defined function is defined:

A The per-call CPU cost of the function.

B The expected percentage of bytes in the argument that the function will read. This factor caters for the situation where a function takes a large object as an argument but may not necessarily use the entire object in its processing.

C The CPU cost per byte read.

The CPU cost of a function invocation is then given by the algorithm A + C* (B * expected size of argument), and the I/O cost is (B * expected size of argument).

Therefore, in an ORDBMS we might expect to be able to provide information to optimize query execution. The problem with this approach is that it can be difficult for a user to provide these figures. An alternative, and more attractive, approach is for the ORDBMS to derive these figures based on experimentation through the handling of functions and objects of differing sizes and complexity.

Example 28.19 Potentially different query processing heuristics

Find all detached properties in Glasgow that are within two miles of a primary school and are managed by Ann Beech.

```
SELECT *
FROM PropertyForRent p, Staff s
WHERE p.staffNo = s.staffNo AND
        p.nearPrimarySchool(p.postcode) < 2.0 AND p.city = 'Glasgow' AND
        s.fName = 'Ann' AND s.lName = 'Beech';
```

For the purposes of this query, we will assume that we have created an external user-defined function nearPrimarySchool, which takes a postcode and determines from an internal database of known buildings (such as residential, commercial, industrial) the distance to the nearest primary school. Translating this to a relational algebra tree, as discussed in Section 21.3, we get the tree shown in Figure 28.3(a). If we now use the general query processing heuristics, we would normally push the Selection operations down past the Cartesian product and transform Cartesian product/Selection into a Join operation, as shown in Figure 28.3(b). In this particular case, this may not be the best strategy. If the user-defined function nearPrimarySchool has a significant amount of processing to perform for each invocation, it may be better to perform the Selection on the Staff table first and then perform the Join operation on staffNo before calling the user-defined function. In this case, we may also use the commutativity of joins rule to rearrange the leaf nodes so that the more restrictive Selection operation is performed first (as the outer relation in a left-deep join tree), as illustrated in Figure 28.3(c). Further, if the query plan for the Selection operation on (nearPrimarySchool() < 2.0 AND city = 'Glasgow') is evaluated in the order given, left to right, and there are no indexes or sort orders defined, then again this is unlikely to be as efficient as first evaluating the Selection operation on (city = 'Glasgow') and then the Selection on (nearPrimarySchool() < 2.0), as illustrated in Figure 28.3(d).

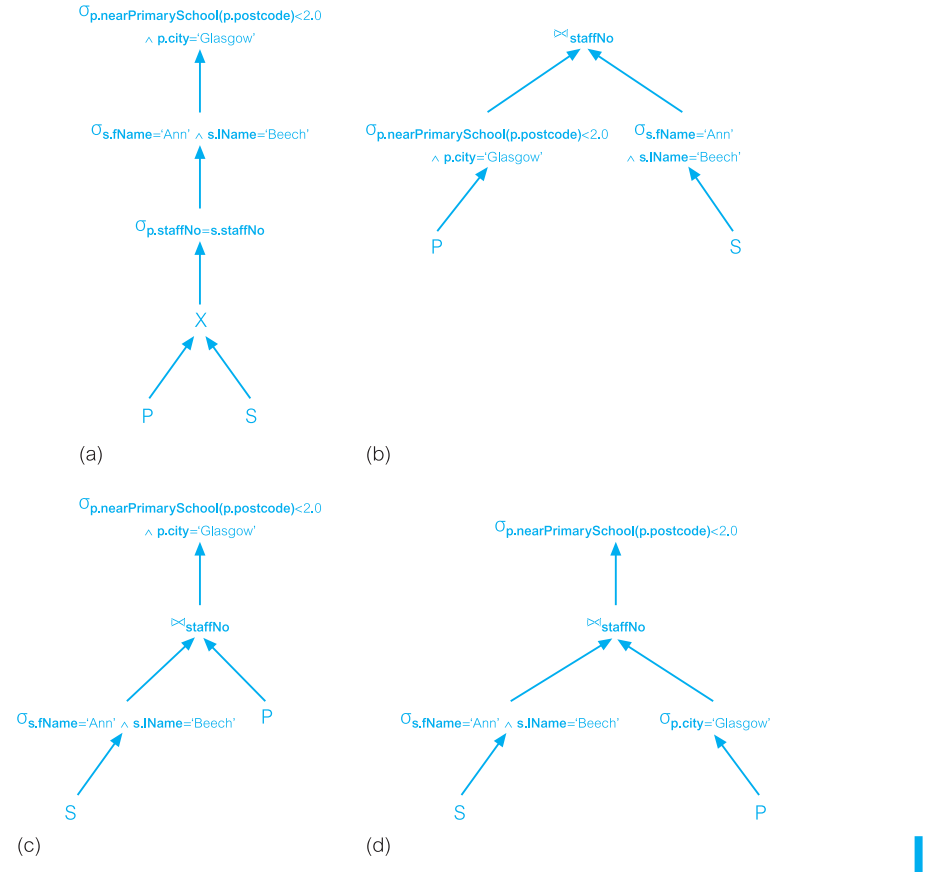


Figure 28.3
(a) Canonical relational algebra tree; (b) optimized relational algebra tree pushing all selections down; (c) optimized relational algebra tree pushing down selection on Staff only; (d) optimized relational algebra tree separating selections on PropertyForRent.

In Example 28.19, the result of the user-defined function nearPrimarySchool is a floating point value that represents the distance between a property and the nearest primary school. An alternative strategy for improving the performance of this query is to add an index, not on the function itself but on the result of the function. For example, in Illustra we can create an index on the result of this UDF using the following SQL statement:

```
CREATE INDEX nearPrimarySchoolIndex
ON PropertyForRent USING B-tree (nearPrimarySchool(postcode));
```

Now whenever a new record is inserted into the PropertyForRent table, or the postcode column of an existing record is updated, the ORDBMS will compute the nearPrimarySchool function and index the result. When a PropertyForRent record is deleted, the ORDBMS will again compute this function to delete the corresponding index record. Consequently, when the UDF appears in a query, Illustra can use the index to retrieve the record and so improve the response time.

Another strategy that should be possible is to allow a UDF to be invoked not from the ORDBMS server, but instead from the client. This may be an appropriate strategy when the amount of processing in the UDF is large, and the client has the power and the ability to execute the UDF (in other words, the client is reasonably heavyweight). This alleviates the processing from the server and helps improve the performance and throughput of the overall system.

This resolves another problem associated with UDFs that we have not yet discussed that has to do with security. If the UDF causes some fatal runtime error, then if the UDF code is linked into the ORDBMS server, the error may have the consequential effect of crashing the server. Clearly, this is something that the ORDBMS has to protect against. One approach is to have all UDFs written in an interpreted language, such as SQL or Java. However, we have already seen that SQL:2003 allows an external routine, written in a high-level programming language such as 'C' or C++, to be invoked as a UDF. In this case, an alternative approach is to run the UDF in a different address space to the ORDBMS server, and for the UDF and server to communicate using some form of inter-process communication (IPC). In this case, if the UDF causes a fatal runtime error, the only process affected is that of the UDF.

New Index Types 28.5.1

In Example 28.19 we saw that it was possible for an ORDBMS to compute and index the result of a user-defined function that returned scalar data (numeric and character data types). Traditional relational DBMSs use B-tree indexes to speed access to scalar data (see Appendix C). However, a B-tree is a one-dimensional access method that is inappropriate for multidimensional access, such as those encountered in geographic information systems, telemetry, and imaging systems. With the ability to define complex data types in an ORDBMS, specialized index structures are required for efficient access to data. Some ORDBMSs are beginning to support additional index types, such as:

- generic B-trees that allow B-trees to be built on any data type, not just alphanumeric;
- quad Trees (Finkel and Bentley, 1974);
- K-D-B Trees (Robinson, 1981).
- R-trees (region trees) for fast access to two- and three-dimensional data (Gutman, 1984);
- grid files (Nievergelt *et al*., 1984);
- D-trees, for text support.

A mechanism to plug in any user-defined index structure provides the highest level of flexibility. This requires the ORDBMS to publish an access method interface that allows users to provide their own access methods appropriate to their particular needs. Although this sounds relatively straightforward, the programmer for the access method has to take account of such DBMS mechanisms as locking, recovery, and page management.

An ORDBMS could provide a generic template index structure that is sufficiently general to encompass most index structures that users might design and interface to the normal DBMS mechanisms. For example, the Generalized Search Tree (GiST) is a template index structure based on B-trees that accommodates many tree-based index structures with minimal coding (Hellerstein *et al.*, 1995).

28.6 Object-Oriented Extensions in Oracle

In Section 8.2 we examined some of the standard facilities of Oracle, including the base data types supported by Oracle, the procedural programming language PL/SQL, stored procedures and functions, and triggers. Many of the object-oriented features that appear in the new SQL:2003 standard appear in Oracle in one form or another. In this section we briefly discuss some of the object-oriented features in Oracle.

28.6.1 User-Defined Data Types

As well as supporting the built-in data types that we discussed in Section 8.2.3, Oracle supports two user-defined data types:

- object types;
- collection types.

Object types

An object type is a schema object that has a name, a set of attributes based on the built-in data types or possibly other object types, and a set of methods, similar to what we discussed for an SQL:2003 object type. For example, we could create Address, Staff, and Branch types as follows:

```
CREATE TYPE AddressType AS OBJECT (
        street      VARCHAR2(25),
        city        VARCHAR2(15),
        postcode    VARCHAR2(8));

CREATE TYPE StaffType AS OBJECT (
        staffNo     VARCHAR2(5),
        fName       VARCHAR2(15),
        lName       VARCHAR2(15),
        position    VARCHAR2(10),
        sex         CHAR,
        DOB         DATE,
        salary      DECIMAL(7, 2),
        MAP MEMBER FUNCTION age RETURN INTEGER,
        PRAGMA RESTRICT_REFERENCES(age, WNDS, WNPS, RNPS))
```

```
NOT FINAL;
CREATE TYPE BranchType AS OBJECT (
        branchNo        VARCHAR2(4),
        address         AddressType,
        MAP MEMBER FUNCTION getbranchNo RETURN VARCHAR2(4),
        PRAGMA RESTRICT_REFERENCES(getbranchNo, WNDS, WNPS,
                                   RNDS, RNPS));
```

We can then create a Branch (object) table using the following statement:

```
CREATE TABLE Branch OF BranchType (branchNo PRIMARY KEY);
```

This creates a Branch table with columns branchNo and address of type AddressType. Each row in the Branch table is an object of type BranchType. The pragma clause is a compiler directive that denies member functions read/write access to database tables and/or package variables (WNDS means *does not modify database tables*, WNPS means *does not modify packaged variables*, RNDS means *does not query database tables*, and RNPS means *does not reference package variables*). This example also illustrates another object-relational feature in Oracle, namely the specification of methods.

Methods

The methods of an object type are classified as member, static, and comparison. A **member** method is a function or a procedure that always has an implicit SELF parameter as its first parameter, whose type is the containing object type. Such methods are useful as observer and mutator functions and are invoked in the *selfish style*, for example object.method(), where the method finds all its arguments among the attributes of the object. We have defined an observer member method getbranchNo in the new type BranchType; we show the implementation of this method shortly.

A **static** method is a function or a procedure that does not have an implicit SELF parameter. Such methods are useful for specifying user-defined constructors or cast methods and may be invoked by qualifying the method with the type name, as in typename.method().

A **comparison** method is used for comparing instances of object types. Oracle provides two ways to define an order relationship among objects of a given type:

- a **map method** uses Oracle's ability to compare built-in types. In our example, we have defined a map method for the new type BranchType, which compares two branch objects based on the values in the branchNo attribute. We show an implementation of this method shortly.
- an **order method** uses its own internal logic to compare two objects of a given object type. It returns a value that encodes the order relationship. For example, it may return −1 if the first is smaller, 0 if they are equal, and 1 if the first is larger.

For an object type, either a map method or an order method can be defined, but not both. If an object type has no comparison method, Oracle cannot determine a greater than or less than relationship between two objects of that type. However, it can attempt to determine whether two objects of the type are equal using the following rules:

- if all the attributes are non-null and equal, the objects are considered equal;
- if there is an attribute for which the two objects have unequal non-null values, the objects are considered unequal;
- otherwise, Oracle reports that the comparison is not available (null).

Methods can be implemented in PL/SQL, Java, and 'C', and overloading is supported provided their formal parameters differ in number, order, or data type. For the previous example, we could create the body for the member functions specified above for types BranchType and StaffType as follows:

```
CREATE OR REPLACE TYPE BODY BranchType AS
  MAP MEMBER FUNCTION getbranchNo RETURN VARCHAR2(4) IS
  BEGIN
        RETURN branchNo;
  END;
END;
CREATE OR REPLACE TYPE BODY StaffType AS
  MAP MEMBER FUNCTION age RETURN INTEGER IS
  var NUMBER;
  BEGIN
        var := TRUNC(MONTHS_BETWEEN(SYSDATE, DOB)/12);
        RETURN var;
  END;
END;
```

The member function getbranchNo acts not only as an observer method to return the value of the branchNo attribute, but also as the comparison (map) method for this type. We see an example of the use of this method shortly. As in SQL:2003, user-defined functions can also be declared separately from the CREATE TYPE statement. In general, user-defined functions can be used in:

- the select list of a SELECT statement;
- a condition in the WHERE clause;
- the ORDER BY or GROUP BY clauses;
- the VALUES clause of an INSERT statement;
- the SET clause of an UPDATE statement.

Oracle also allows user-defined operators to be created using the CREATE OPERATOR statement. Like built-in operators, a user-defined operator takes a set of operands as input and return a result. Once a new operator has been defined, it can be used in SQL statements like any other built-in operator.

Constructor methods

Every object type has a system-defined *constructor method* that makes a new object according to the object type's specification. The constructor method has the same name as the object type and has parameters that have the same names and types as the object type's attributes. For example, to create a new instance of BranchType, we could use the following expression:

BranchType('B003', AddressType('163 Main St', 'Glasgow', 'G11 9QX'));

Note, the expression AddressType('163 Main St', 'Glasgow', 'G11 9QX') is itself an invocation of the constructor for the type AddressType.

Object identifiers

Every row object in an object table has an associated logical object identifier (OID), which by default is a unique system-generated identifier assigned for each row object. The purpose of the OID is to uniquely identify each row object in an object table. To do this, Oracle implicitly creates and maintains an index on the OID column of the object table. The OID column is hidden from users and there is no access to its internal structure. While OID values in themselves are not very meaningful, the OIDs can be used to fetch and navigate objects. (Note, objects that appear in object tables are called *row objects* and objects that occupy columns of relational tables or as attributes of other objects are called *column objects*.)

Oracle requires every row object to have a unique OID. The unique OID value may be specified to come from the row object's primary key or to be system-generated, using either the clause OBJECT IDENTIFIER PRIMARY KEY or OBJECT IDENTIFIER SYSTEM GENERATED (the default) in the CREATE TABLE statement. For example, we could restate the creation of the Branch table as:

```
CREATE TABLE Branch OF BranchType (branchNo PRIMARY KEY)
OBJECT IDENTIFIER PRIMARY KEY;
```

REF data type

Oracle provides a built-in data type called REF to encapsulate references to row objects of a specified object type. In effect, a REF is used to model an association between two row objects. A REF can be used to examine or update the object it refers to and to obtain a copy of the object it refers to. The only changes that can be made to a REF are to replace its contents with a reference to a different object of the same object type or to assign it a null value. At an implementation level, Oracle uses object identifiers to construct REFs.

As in SQL:2003, a REF can be constrained to contain only references to a specified object table, using a SCOPE clause. As it is possible for the object identified by a REF to become unavailable, for example through deletion of the object, Oracle SQL has a predicate IS DANGLING to test REFs for this condition. Oracle also provides a dereferencing operator, DEREF, to access the object referred to by a REF. For example, to model the manager of a branch we could change the definition of type BranchType to:

```
CREATE TYPE BranchType AS OBJECT (
        branchNo        VARCHAR2(4),
        address         AddressType,
        manager         REF StaffType,
        MAP MEMBER FUNCTION getbranchNo RETURN VARCHAR2(4),
        PRAGMA RESTRICT_REFERENCES(getbranchNo, WNDS, WNPS,
                                   RNDS, RNPS));
```

In this case, we have modeled the manager through the reference type, REF StaffType. We see an example of how to access this column shortly.

Type inheritance

Oracle supports single inheritance allowing a subtype to be derived from a single parent type. The subtype inherits all the attributes and methods of the supertype and additionally can add new attributes and methods, and it can override any of the inherited methods. As with SQL:2003, the UNDER clause is used to specify the supertype.

Collection types

Oracle currently supports two collection types: array types and nested tables.

Array types

An **array** is an ordered set of data elements that are all of the same data type. Each element has an **index**, which is a number corresponding to the element's position in the array. An array can have a fixed or variable size, although in the latter case a maximum size must be specified when the array type is declared. For example, a branch office can have up to three telephone numbers, which we could model in Oracle by declaring the following new type:

```
CREATE TYPE TelNoArrayType AS VARRAY(3) OF VARCHAR2(13);
```

The creation of an array type does not allocate space but rather defines a data type that can be used as:

- the data type of a column of a relational table;
- an object type attribute;
- a PL/SQL variable, parameter, or function return type.

For example, we could modify the type BranchType to include an attribute of this new type:

```
phoneList       TelNoArrayType,
```

An array is normally stored inline, that is, in the same tablespace as the other data in its row. If it is sufficiently large, however, Oracle stores it as a BLOB.

Nested tables

A **nested table** is an unordered set of data elements that are all of the same data type. It has a single column of a built-in type or an object type. If the column is an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type. For example, to model next-of-kin for members of staff, we may define a new type as follows:

```
CREATE TYPE NextOfKinType AS OBJECT (
        fName       VARCHAR2(15),
        lName       VARCHAR2(15),
        telNo       VARCHAR2(13));
CREATE TYPE NextOfKinNestedType AS TABLE OF NextOfKinType;
```

We can now modify the type StaffType to include this new type as a nested table:

```
nextOfKin       NextOfKinNestedType,
```

and create a table for staff using the following statement:

```
CREATE TABLE Staff OF StaffType (
        PRIMARY KEY staffNo)
        OBJECT IDENTIFIER IS PRIMARY KEY
        NESTED TABLE nextOfKin STORE AS NextOfKinStorageTable (
                (PRIMARY KEY(Nested_Table_Id, lName, telNo))
                 ORGANIZATION INDEX COMPRESS)
        RETURN AS LOCATOR;
```

The rows of a nested table are stored in a separate storage table that cannot be directly queried by the user but can be referenced in DDL statements for maintenance purposes. A hidden column in this storage table, Nested_Table_Id, matches the rows with their corresponding parent row. All the elements of the nested table of a given row of Staff have the same value of Nested_Table_Id and elements that belong to a different row of Staff have a different value of Nested_Table_Id.

We have indicated that the rows of the nextOfKin nested table are to be stored in a separate storage table called NextOfKinStorageTable. In the STORE AS clause we have also specified that the storage table is index-organized (ORGANIZATION INDEX), to cluster rows belonging to the same parent. We have specified COMPRESS so that the Nested_Table_Id part of the index key is stored only once for each row of a parent row rather than being repeated for every row of a parent row object.

The specification of Nested_Table_Id and the given attributes as the primary key for the storage table serves two purposes: it serves as the key for the index and it enforces uniqueness of the columns (lName, telNo) of a nested table within each row of the parent table. By including these columns in the key, the statement ensures that the columns contain distinct values within each member of staff.

In Oracle, the collection typed value is encapsulated. Consequently, a user must access the contents of a collection via interfaces provided by Oracle. Generally, when the user accesses a nested table, Oracle returns the entire collection value to the user's client process. This may have performance implications, and so Oracle supports the ability to return a nested table value as a locator, which is like a handle to the collection value. The RETURN AS LOCATOR clause indicates that the nested table is to be returned in the locator form when retrieved. If this is not specified, the default is VALUE, which indicates that the entire nested table is to be returned instead of just a locator to the nested table.

Nested tables differ from arrays in the following ways:

- Arrays have a maximum size, but nested tables do not.
- Arrays are always dense, but nested tables can be sparse, and so individual elements can be deleted from a nested table but not from an array.
- Oracle stores array data in-line (in the same tablespace) but stores nested table data out-of-line in a *store table*, which is a system-generated database table associated with the nested table.
- When stored in the database, arrays retain their ordering and subscripts, but nested tables do not.

28.6.2 Manipulating Object Tables

In this section we briefly discuss how to manipulate object tables using the sample objects created above for illustration. For example, we can insert objects into the Staff table as follows:

```
INSERT INTO Staff VALUES ('SG37', 'Ann', 'Beech', 'Assistant', 'F',
  '10-Nov-1960', 12000, NextOfKinNestedType());
INSERT INTO Staff VALUES ('SG5', 'Susan', 'Brand', 'Manager', 'F',
  '3-Jun-1940', 24000, NextOfKinNestedType());
```

The expression NextOfKinNestedType() invokes the constructor method for this type to create an empty nextOfKin attribute. We can insert data into the nested table using the following statement:

```
INSERT INTO TABLE (SELECT s.nextOfKin
                   FROM Staff s
                   WHERE s.staffNo = 'SG5')
VALUES ('John', 'Brand', '0141-848-2000');
```

This statement uses a TABLE expression to identify the nested table as the target for the insertion, namely the nested table in the nextOfKin column of the row object in the Staff table that has a staffNo of 'SG5'. Finally, we can insert an object into the Branch table:

```
INSERT INTO Branch
  SELECT 'B003', AddressType('163 Main St', 'Glasgow', 'G11 9QX'), REF(s),
                 TelNoArrayType('0141-339-2178', '0141-339-4439')
  FROM Staff s
  WHERE s.staffNo = 'SG5';
```

or alternatively:

```
INSERT INTO Branch VALUES ('B003', AddressType('163 Main St', 'Glasgow',
  'G11 9QX'), (SELECT REF(s) FROM Staff s WHERE s.staffNo = 'SG5'),
  TelNoArrayType('0141-339-2178', '0141-339-4439'));
```

Querying object tables

In Oracle, we can return an ordered list of branch numbers using the following query:

```
SELECT b.branchNo
FROM Branch b
ORDER BY VALUE(b);
```

This query implicitly invokes the comparison method getbranchNo that we defined as a map method for the type BranchType to order the data in ascending order of branchNo. We can return all the data for each branch using the following query:

```
SELECT b.branchNo, b.address, DEREF(b.manager), b.phoneList
FROM Branch b
WHERE b.address.city = 'Glasgow'
ORDER BY VALUE(b);
```

Note the use of the DEREF operator to access the manager object. This query writes out the values for the branchNo column, all columns of an address, all columns of the manager object (of type StaffType), and all relevant telephone numbers.

We can retrieve next of kin data for all staff at a specified branch using the following query:

```
SELECT b.branchNo, b.manager.staffNo, n.*
FROM Branch b, TABLE(b.manager.nextOfKin) n
WHERE b.branchNo = 'B003';
```

Many applications are unable to handle collection types and instead require a flattened view of the data. In this example, we have flattened (or *unnested*) the nested set using the TABLE keyword. Note also that the expression b.manager.staffNo is a shorthand notation for y.staffNo where y = **DEREF**(b.manager).

Object Views 28.6.3

In Sections 3.4 and 6.4 we examined the concept of views. In much the same way that a view is a virtual table, an **object view** is a virtual object table. Object views allow the data to be customized for different users. For example, we may create a view of the Staff table to prevent some users from seeing sensitive personal or salary-related information. In Oracle, we may now create an object view that not only restricts access to some data but also prevents some methods from being invoked, such as a delete method. It has also been argued that object views provide a simple migration path from a purely relational-based application to an object-oriented one, thereby allowing companies to experiment with this new technology.

For example, assume that we have created the object types defined in Section 28.6.1 and assume that we have created and populated the following relational schema for *DreamHome* with associated structured types BranchType and StaffType:

Branch	(branchNo, street, city, postcode, mgrStaffNo)
Telephone	(telNo, branchNo)
Staff	(staffNo, fName, lName, position, sex, DOB, salary, branchNo)
NextOfKin	(staffNo, fName, lName, telNo)

We could create an object-relational schema using the object view mechanism as follows:

```
CREATE VIEW StaffView OF StaffType WITH OBJECT IDENTIFIER (staffNo) AS
  SELECT s.staffNo, s.fName, s.lName, s.sex, s.position, s.DOB, s.salary,
         CAST (MULTISET (SELECT n.fName, n.lName, n.telNo
           FROM NextOfKin n WHERE n.staffNo = s.staffNo)
           AS NextOfKinNestedType) AS nextOfKin
  FROM Staff s;
CREATE VIEW BranchView OF BranchType WITH OBJECT IDENTIFIER
(branchNo) AS
  SELECT b.branchNo, AddressType(b.street, b.city, b.postcode) AS address,
         MAKE_REF(StaffView, b.mgrStaffNo) AS manager,
```

```
        CAST (MULTISET (SELECT telNo FROM Telephone t
            WHERE t.branchNo = b.branchNo) AS TelNoArrayType) AS phoneList
    FROM Branch b;
```

In each case, the SELECT subquery inside the CAST/MULTISET expression selects the data we require (in the first case, a list of next of kin for the member of staff and in the second case, a list of telephone numbers for the branch). The MULTISET keyword indicates that this is a list rather than a singleton value, and the CAST operator then casts this list to the required type. Note also the use of the MAKE_REF operator, which creates a REF to a row of an object view or a row in an object table whose object identifier is primary-key based.

The WITH OBJECT IDENTIFIER specifies the attributes of the object type that will be used as a key to identify each row in the object view. In most cases, these attributes correspond to the primary key columns of the base table. The specified attributes must be unique and identify exactly one row in the view. If the object view is defined on an object table or an object view, this clause can be omitted or WITH OBJECT IDENTIFIER DEFAULT can be specified. In each case, we have specified the primary key of the corresponding base table to provide uniqueness.

28.6.4 Privileges

Oracle defines the following system privileges for user-defined types:

- CREATE TYPE – to create user-defined types in the user's schema;
- CREATE ANY TYPE – to create user-defined types in any schema;
- ALTER ANY TYPE – to alter user-defined types in any schema;
- DROP ANY TYPE – to drop named types in any schema;
- EXECUTE ANY TYPE – to use and reference named types in any schema.

In addition, the EXECUTE schema object privilege allows a user to use the type to define a table, define a column in a relational table, declare a variable or parameter of the named type, and to invoke the type's methods.

28.7 Comparison of ORDBMS and OODBMS

We conclude our treatment of object-relational DBMSs and object-oriented DBMSs with a brief comparison of the two types of system. For the purposes of the comparison, we examine the systems from three perspectives: data modeling (Table 28.2), data access (Table 28.3), and data sharing (Table 28.4). We assume that future ORDBMSs will be compliant with the SQL:1999/2003 standard.

Table 28.2 Data modeling comparison of ORDBMS and OODBMS.

Feature	ORDBMS	OODBMS
Object identity (OID)	Supported through REF type	Supported
Encapsulation	Supported through UDTs	Supported but broken for queries
Inheritance	Supported (separate hierarchies for UDTs and tables)	Supported
Polymorphism	Supported (UDF invocation based on the generic function)	Supported as in an object-oriented programming model language
Complex objects	Supported through UDTs	Supported
Relationships	Strong support with user-defined referential integrity constraints	Supported (for example, using class libraries)

Table 28.3 Data access comparison of ORDBMS and OODBMS.

Feature	ORDBMS	OODBMS
Creating and accessing persistent data	Supported but not transparent	Supported but degree of transparency differs between products
Ad hoc query facility	Strong support	Supported through ODMG 3.0
Navigation	Supported by REF type	Strong support
Integrity constraints	Strong support	No support
Object server/page server	Object server	Either
Schema evolution	Limited support	Supported but degree of support differs between products

Table 28.4 Data sharing comparison of ORDBMS and OODBMS.

Feature	ORDBMS	OODBMS
ACID transactions	Strong support	Supported
Recovery	Strong support	Supported but degree of support differs between products
Advanced transaction models	No support	Supported but degree of support differs between products
Security, integrity, and views	Strong support	Limited support

Chapter Summary

- There is no single extended relational data model; rather, there are a variety of these models, whose characteristics depend upon the way and the degree to which extensions were made. However, all the models do share the same basic relational tables and query language, all incorporate some concept of 'object', and some have the ability to store methods or procedures/triggers as well as data in the database.
- Various terms have been used for systems that have extended the relational data model. The original term used to describe such systems was the *Extended Relational DBMS* (ERDBMS). However, in recent years, the more descriptive term **Object-Relational DBMS** (ORDBMS) has been used to indicate that the system incorporates some notion of 'object', and the term *Universal Server* or *Universal DBMS* (UDBMS) has also been used.
- SQL:1999 and SQL:2003 extensions include: row types, user-defined types (UDTs) and user-defined routines (UDRs), polymorphism, inheritance, reference types and object identity, collection types (ARRAYs), new language constructs that make SQL computationally complete, triggers, and support for large objects – Binary Large Objects (BLOBs) and Character Large Objects (CLOBs) – and recursion.
- The query optimizer is the heart of RDBMS performance and must also be extended with knowledge about how to execute user-defined functions efficiently, take advantage of new index structures, transform queries in new ways, and navigate among data using references. Successfully opening up such a critical and highly tuned DBMS component and educating third parties about optimization techniques is a major challenge for DBMS vendors.
- Traditional RDBMSs use B-tree indexes to speed access to scalar data. With the ability to define complex data types in an ORDBMS, specialized index structures are required for efficient access to data. Some ORDBMSs are beginning to support additional index types, such as generic B-trees, R-trees (region trees) for fast access to two- and three-dimensional data, and the ability to index on the output of a function. A mechanism to plug in any user-defined index structure provides the highest level of flexibility.

Review Questions

28.1 What functionality would typically be provided by an ORDBMS?

28.2 What are the advantages and disadvantages of extending the relational data model?

28.3 What are the main features of the SQL:2003 standard?

28.4 Discuss how references types and object identity can be used.

28.5 Compare and contrast procedures, functions, and methods.

28.6 What is a trigger? Provide an example of a trigger.

28.7 Discuss the collection types available in SQL:2003.

28.8 Discuss how SQL:2003 supports recursive queries. Provide an example of a recursive query in SQL.

28.9 Discuss the extensions required to query processing and query optimization to fully support the ORDBMS.

28.10 What are the security problems associated with the introduction of user-defined methods? Suggest some solutions to these problems.

Exercises

28.11 Analyze the RDBMSs that you are currently using. Discuss the object-oriented facilities provided by the system. What additional functionality do these facilities provide?

28.12 Consider the relational schema for the Hotel case study given in the Exercises at the end of Chapter 3. Redesign this schema to take advantage of the new features of SQL:2003. Add user-defined functions that you consider appropriate.

28.13 Create SQL:2003 statements for the queries given in Chapter 5, Exercises 5.7–5.28.

28.14 Create an insert trigger that sets up a mailshot table recording the names and addresses of all guests who have stayed at the hotel during the days before and after New Year for the past two years.

28.15 Repeat Exercise 28.7 for the multinational engineering case study in the Exercises of Chapter 22.

28.16 Create an object-relational schema for the *DreamHome* case study documented in Appendix A. Add user-defined functions that you consider appropriate. Implement the queries listed in Appendix A using SQL:2003.

28.17 Create an object-relational schema for the *University Accommodation Office* case study documented in Appendix B.1. Add user-defined functions that you consider appropriate.

28.18 Create an object-relational schema for the *EasyDrive School of Motoring* case study documented in Appendix B.2. Add user-defined functions that you consider appropriate.

28.19 Create an object-relational schema for the *Wellmeadows* case study documented in Appendix B.3. Add user-defined functions that you consider appropriate.

28.20 You have been asked by the Managing Director of *DreamHome* to investigate and prepare a report on the applicability of an object-relational DBMS for the organization. The report should compare the technology of the RDBMS with that of the ORDBMS, and should address the advantages and disadvantages of implementing an ORDBMS within the organization, and any perceived problem areas. The report should also consider the applicability of an object-oriented DBMS, and a comparison of the two types of system for *DreamHome* should be included. Finally, the report should contain a fully justified set of conclusions on the applicability of the ORDBMS for *DreamHome*.