OBJECT-ORIENTED DATABASES

Traditional database management systems (DBMS), based on the relational data model, are not able to directly handle data managed by a large variety of applications, such as design and manufacturing systems (CAD/CAM, CIM), scientific and medical databases, geographic information systems, and multimedia databases. Those applications have requirements and characteristics different from those typical of traditional database applications for business and administration. They are characterized by highly structured data, long transactions, and data types for storing images and texts, as well as by nonstandard, application-specific operations. Object-oriented database management systems (OODBMS) (1–3) have been developed in order to meet the requirements imposed by those applications. The object-oriented approach provides the required flexibility not being constrained by the data types and query languages available in traditional database systems. One of the most important features of OODBMS is the possibility they give to the applications of specifying both the structures of complex objects and the operations to manipulate these structures.

OODBMS result from the integration of database technology with the object-oriented paradigm developed in the programming languages and software engineering areas. The basic principle of the object-oriented approach in programming is to consider the program consisting of independent objects, grouped in classes, communicating among each other through messages. The same concepts, however, have been introduced also in other computer science areas, such as knowledge representation languages, and they have often been interpreted in different ways.

In an object-oriented programming language, objects exist only during program execution. In an object-oriented database, by contrast, objects can be created that persist and can be shared by several programs. Thus, object-oriented databases store persistent objects in secondary memory and support object sharing among different applications. This requires the integration of the object-oriented paradigm with typical DBMS mechanisms such as indexing mechanisms, concurrency control, and transaction management mechanisms.

The history of OODBMS has been characterized by an initial stage of strong development activity, with the realization of many prototype and commercial systems. The first systems were released at the end of the 1980s, and many commercial products were already available at the beginning of the 1990s. Only at a second stage was the need felt for formal foundations and standardization. Thus, the definition of a standard object-oriented data model [by the Object Data Management Group (ODMG)] is quite recent. In the same time, there has been an evolution from the first systems, which mainly were persistent versions of object-oriented programming languages, toward the full support of typical DBMS features, such as declarative query languages, concurrency control, and authorization mechanisms. At the current stage, the field of OODBMS is rather mature, with a standard data model and query language having been defined and with several commercial products available. A different evolutive direction that has been taken, starting from traditional relational DBMS and which is now converging with the one taken by OODBMS, is that of object-relational database systems, that is, object extensions of relational database systems. The latest proposed version of the SQL standard, SQL3, indeed includes many features of the object paradigm.

In this article, we first briefly introduce the notions and the advantages of the object-oriented paradigm in software development, and then we discuss in detail the application of that paradigm to the database context, focusing on data model and query language aspects. After having introduced these notions, we examine some OODBMS; in particular, we discuss the GemStone and ObjectStore systems. The ODMG standard is then presented; we discuss its data model and its query language. Finally, we briefly discuss object-relational databases and compare them with object-oriented ones.

THE OBJECT-ORIENTED PARADIGM

Most of the principles underlying the object-oriented programming paradigm date back to the Simula language (4); however, this paradigm started to be widely used in the following years, mainly because of the development of the Smalltalk (5) language. Many object-oriented programming languages have been proposed, namely, Eiffel (6), CLOS (7), C++ (8), Java (9).

The key concepts of the object-oriented paradigm are those of *object, class, inheritance, encapsulation,* and *polymorphism* (10). Each real-world entity is modeled as an object. An object has an identifier (OID), a state, and a set of operations. The effect of the execution of an operation on an object depends on both the object state and the operation arguments and can result in an update of the object state.

Classify group objects with similar characteristics—for example, all the objects answering the same set of messages. A class is also a template from which objects can be created, through a new operation. Objects belonging to the same class have the same operations and thus they exhibit a uniform behavior. Classes have an interface, specifying the operations that can be invoked on objects belonging to the class, and an implementation, specifying the code implementing the operations in the class interface.

Inheritance allows a class to be defined starting from the definitions of existing classes. A class can be defined as a specialization of one or more existing classes and thus can inherit attributes and methods of those classes. The class defined as a specialization is called a subclass, whereas the classes from which it is derived are called superclasses. An object can use operations defined in its base class as well as in its superclasses. Inheritance is thus a powerful mechanism for code reuse.

Encapsulation allows us to hide data representation and operation implementation. Each object encloses both the procedures (operations, or methods) and the interface through which the object can be accessed and modified by other objects; the object interface consists of the set of operations that can be invoked on the object. An object state can be manipulated only through the execution of object methods.

Polymorphism (overloading) allows us to define operations with the same name for different object types; together with overriding, that is, the possibility of redefining implementations of inherited methods in subclasses, and late binding, this functionality allows an operation to behave differently on objects of different classes. Different methods can thus be associated with the same operation name; and the task to decide, at execution time, which method to use for executing a given operation, is left to the system.

The impact of the above concepts on programming methodologies is relevant. Objects encapsulate operations together with the data these operations modify, thus providing a data-oriented approach to program development. Objects are dealt with as first class values in the language, and thus they can be passed as parameters and can be assigned as values to variables and organized in structures. Classes simplify handling collections of similar objects. Finally, inheritance among classes is a mechanism to organize collections of classes, thus allowing the application domain to be described by class hierarchies.

The great popularity of the object-oriented approach in software development is mainly due to increased productivity. With respect to the software life cycle, the object-oriented paradigm reduces time on two different sides: On one side, the development time is reduced, because of specification and implementation reuse; on the other side, the maintenance cost is reduced, because of the locality of modifications. The object-oriented paradigm enhances software reusability and extensibility. It reduces the amount of code to be written and makes the design faster through reuse. This paradigm can be seen as a collection of methods and tools for structuring software. In this respect, class libraries have a fundamental relevance.

A class library is a set of related classes concerning a specific domain. Class libraries can be bought, in the case of standard base modules, or they can be developed in house, if application-specific. The style of programming based on reusable modules, besides improving productivity, also improves quality and testability and makes it easier to modify a program.

A further advantage of object orientation is represented by the uniqueness of the paradigm. In the traditional software life cycle, many barriers should be overcome when passing from the real world (the problem domain) to analysis (e.g., structured analysis or DFD), to programming (e.g., in Fortran, C, or Cobol), and finally to databases [usually relational ones, and designed through the Entity–Relationship approach (11)]. Each of these steps introduces some communication problems. In the object-oriented software life cycle, by contrast, all the various phases (analysis, design, programming, and so on) rely on the same model, and thus the transition from one phase to another is smooth and natural. Requirement analysis and validation is also easier. By using an object-oriented database system, moreover, the problem of type system mismatch between the DBMS and the programming language, known as "impedance mismatch," is overcome, and there is no longer the need for separately designing the database structure.

Finally, the object-oriented paradigm represents a fundamental shift with respect to how the software is produced: The software is no longer organized according to the computer execution model (in a procedural way); rather it is organized according to the human way of thinking. This makes the analysis and design of software systems easier, by allowing the user to participate in the analysis and design. The object-oriented paradigm, being based on the human view of the world, overcomes the communication difficulties often arising between the system analyst and the domain expert.

The object-oriented technology offers some other advantages with respect to analysis and design. It improves the internal coherence of analysis results by integrating data and operations on them. The inheritance mechanism naturally supports the decomposition of problems in subproblems, thus facilitating the handling of complex problems by identifying common subproblems.

Finally, the object-oriented paradigm is well-suited for heterogeneous system integration, which is required in many applications. An important requirement is that new applications be able to (a) interact with existing ones and (b) access the data handled by those applications. This requirement is crucial since the development of computerized information systems usually goes through several phases. Very often, the choice of a specific programming language or of a specific DBMS depends on current requirements of the application or on the available technology. Since both those factors vary over time, organizations are frequently forced to use heterogeneous systems, which are often of different types and thus interconnection problems arise. There is a growing interest in the possibility of exploiting the object-oriented approach to integrate heterogeneous systems. The object-oriented paradigm itself, because of encapsulation, promises to be the most natural approach to solve the integration problems not yet solved by traditional approaches.

OBJECT-ORIENTED DATA MODELS

Research in the area of object-oriented databases has been characterized by a strong experimental work and the development of several prototype systems, whereas only later theoretical foundations have been investigated and standards have been developed. In what follows, we introduce the main concepts of the object-oriented data model. We then discuss two specific systems (GemStone and ObjectStore) to better illustrate the object-oriented data model. Finally, we present the recently proposed ODMG standard.

Objects

An object-oriented database is a collection of objects. In object-oriented systems, each real-world entity is represented by an object. Each object has a state and a behavior. The state consists of the values of the object attributes; the behavior is specified by the methods that act on the object state. One of the most important properties of an object is that of having an identity, different from the identity of any other object and immutable during the object lifetime.

Many OODBMS actually do not require each entity to be represented as an object; rather they distinguish between objects and values. The differences between values and objects are the following (12):

- Values are universally known abstractions, and they have the same meaning for each user; objects, by contrast, correspond to abstractions whose meaning is specified in the context of the application.
- Values are *built-in* in the system and do not need to be defined; objects, by contrast, must be introduced in the system through a definition.
- The information represented by a value is the value itself, whereas the meaningful information represented by an object is given by the relationships it has with other objects and values; values are therefore used to describe other entities, whereas objects are the entities being described.

Thus, values are elements of built-in domains, whereas objects are elements of uninterpreted domains. Typical examples of values are: integers, reals, strings. Each object is assigned an immutable identifier, whereas a value has no identifier (rather it is identified by itself).

Object Identity. Each object is uniquely identified by an object identifier (OID), providing it with an identity independent from its value. The OID is unique within the system and is immutable; that is, it does not depend on the state of the object. Object identifiers are usually not directly visible and accessible by the database users; rather they are internally used by the system to identify objects and to support object references through object attribute values. Objects can thus be interconnected and can share components. The semantics of object sharing is illustrated in Fig. 1. The figure shows two objects that, in case (b), share a component, whereas in case (a) these objects do not share any component and simply have the same value for the attribute `date`. While in case (a) a change in the publication date of `Article[i]` from `March 1997` to `April 1997` does not affect the publication date of `Article[j]`, in case (b) the change is also reflected on `Article[j]`.

The notion of object identifier is quite different from the notion of *key* used in the relational model to uniquely identify each tuple in a relation. A key is defined as the value of one

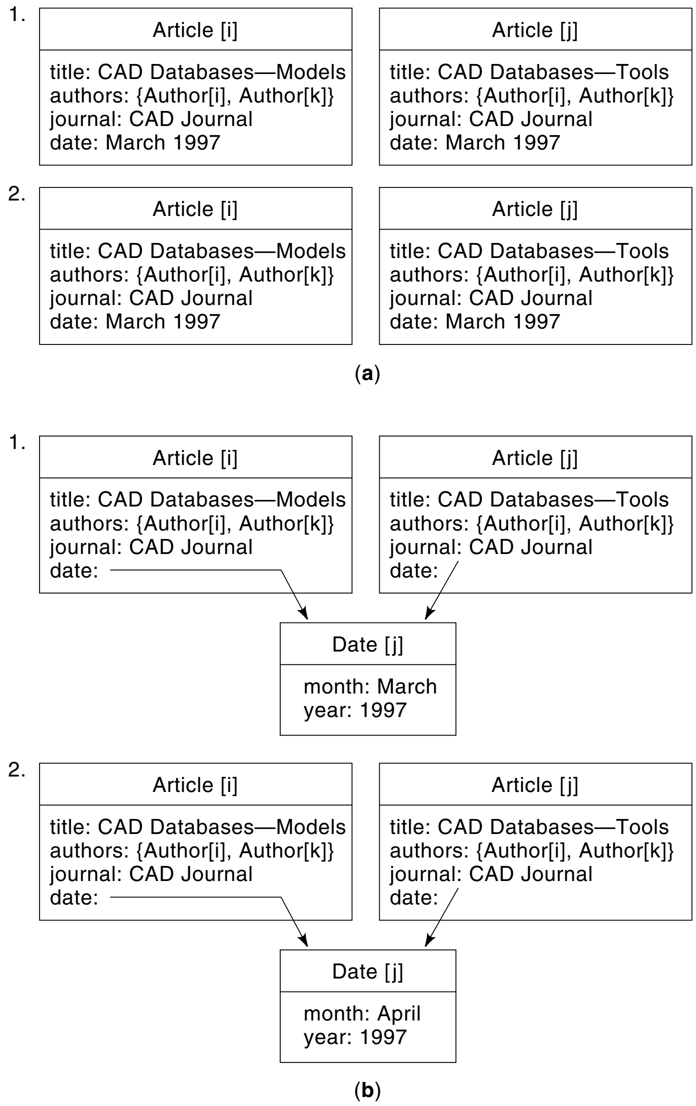

Figure 1. Object-sharing semantics: in case (a) the two objects have the same value for attribute date, whereas in case (b) they share a component.

or more attributes, and it can be modified, whereas an OID is independent from the value of an object state. In particular, two different objects have different OIDs even when their attributes have the same values. Moreover, a key is unique with respect to a relation, whereas an OID is unique within the entire database. The use of OIDs as an identification mechanism has a number of advantages with respect to the use of keys. First of all, because the OIDs are implemented by the system, the application programmer does not have to select the appropriate keys for the various sets of objects. Moreover, because the OIDs are implemented at a low level by the system, better performance is achieved. A disadvantage in the use of OIDs with respect to keys could be the fact that no semantic meaning is associated with them. Note, however, that very often in relational systems, for efficiency reasons, users adopt semantically meaningless codes as keys, especially when foreign keys need to be used.

The notion of object identity introduces at least two different notions of object equality:

- **Equality by identity:** Two objects are identical if they are the same object—that is, if they have the same identifier.
- **Equality by value:** Two objects are equal if the values for their attributes are recursively equal.

Obviously, two identical objects are also equal, whereas the converse does not hold. Figure 2 shows an example of objects which are equal but not identical. Some object-oriented data models also provide a third kind of equality, known as shallow value equality by which two objects are equal, though not being identical, if they share all attributes.

Object Structure. In an object-oriented database the value associated with an object (that is, its state) is a complex value which can be built starting from other objects and values, using some type constructors. Complex (or structured) values are obtained by applying those constructors to simpler objects and values. Examples of primitive values are integers, characters, strings, booleans, and reals. The minimal set of constructors that a system should provide include sets, lists, and tuples. In particular, sets are crucial since they are a natural way to represent real-world collections and multivalued attributes; the tuple constructor is important since it provides a natural way to represent the properties of an entity; lists and arrays are similar to sets, but they impose an order on the elements of the collection and are needed in many scientific applications. Those constructors can be arbitrarily nested. A complex value can contain as components (references to) objects.

Object-oriented databases thus provide an extensible type system that enables the users to define new types, according to the requirements of the applications. The types provided by the system and those defined by the users can be used exactly in the same way.

Many OODBMSs support storage and retrieval of nonstructured values of large size, such as character strings or bit strings. Those values are passed as they are—that is, without being interpreted—to the application program for the interpretation. Those values, which are known as BLOBs (binary large objects), are big-sized values like image bitmaps or long text strings. Those values are not structured in that the DBMS does not know their structure; rather the application using them knows as to interpret them. For example, the ap-

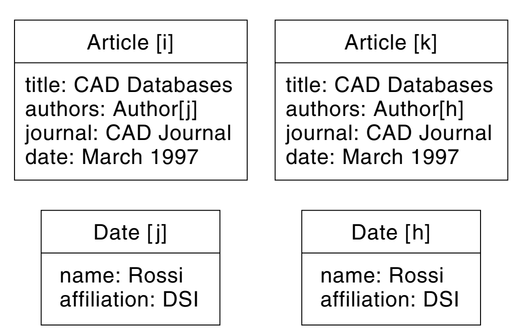

Figure 2. An example of equal, but not identical objects. They have the same state, though different identifiers.

plication may contain some functions to display an image or to search for some keywords in a text.

Methods. Objects in an object-oriented database are manipulated through methods. A method definition usually consists of two components: a signature and an implementation. The signature specifies the method name, the names and types of method arguments, and the type of the result for methods returning a result value. Thus, the signature is a specification of the operation implemented by the method. Some OODBMS do not require the specification of argument types; however, this specification is required in systems performing static type checking. The method implementation consists of a set of instructions expressed in a programming language. Various OODBMS exploit different languages. For instance, ORION exploits Lisp; GemStone exploits a Smalltalk extension, namely OPAL; and O_2 exploits a C extension, namely O_2C. Other systems, including ObjectStore and Ode, exploit C++.

The use of a general-purpose computationally complete programming language to code methods allows the whole application to be expressed in terms of objects. Thus there is no longer the need, typical of relational DBMSs, of embedding the query language (e.g., SQL) in a programming language.

Encapsulation. In a relational DBMS, queries and application programs acting on relations are usually expressed in an imperative language incorporating statements of the data manipulation language (DML) and are stored in a traditional file system rather than in the database. In such an approach, therefore, there is a sharp distinction between programs and data and between query language and programming language. In an object-oriented database, as well as operations manipulating them, are encapsulated in a single structure: the object. Data and operations are thus designed together and they are both stored in the same system. Encapsulation thus provides a sort of "logical data independence," allowing modifications on the data without requiring modifications to the applications using the data.

The notion of encapsulation in programming languages derives from the concept of abstract data type. In this view, an object consists of an interface and an implementation. The interface is the specification of the operations that can be executed on the object, and they are the only part of the object that can be seen from outside. Implementation, by contrast, contains data—that is, the representation or state of the object—and methods specifying the implementation of each operation. This principle, in the database context, is reflected in the fact that an object contains both programs and data, with a variation: In the database context it is not clear whether or not the structure defining the type of an object is part of the interface. In the programming language context, the data structure is usually part of the implementation and, thus, is not visible. For example, in a programming language the data type `list` should be independent from the fact that lists are implemented as arrays or as dynamic structures, and thus this information is correctly hidden. By contrast, in the database context, the knowledge of class attributes, and references made through them to other classes, is often useful.

Some OODBMS, like ORION, allow us to read and write the object attribute values, thus violating encapsulation. The reason for that is to simplify the development of applications that simply access and modify object attributes. Those applications are obviously very common in the database context. Strict encapsulation would require writing many trivial methods. Other systems, like O_2, allow us to specify which methods and attributes are visible in the object interface and thus can be invoked from outside the object. Those attributes and methods are called public, whereas those that cannot be seen from outside the object are called private. Finally, some other systems, including GemStone, force strict encapsulation.

Classes

Instantiation is the mechanism offering the possibility of exploiting the same definition to generate objects with the same structure and behavior. Object-oriented languages provide the notion of class as a basis for instantiation. In this respect a class acts as a template, by specifying:

- A structure—that is, the set of instance attributes
- A set of operations defining the instance interface
- A set of methods implementing the operations

Given a class, the new operation generates objects on which all methods defined for the class can be executed. Obviously, the attribute values must be stored separately for each object; however, there is no need to replicate method definitions, which are associated with the class.

There are, however, some class features that cannot be seen as attributes of its instances, such as the number of class instances present in each moment in the database or the average value of an attribute. An example of an operation which is invoked on classes rather than on objects is the new operation for creating new instances. Some object-oriented data models, like those of GemStone and ORION, allow the definition of attributes and methods characterizing the class as an object, which are therefore not inherited by the class instances.

In almost all object-oriented data models, each attribute has a domain specifying the class of possible objects that can be assigned as values to the attribute. If an attribute of a class C has a class C' as domain, each C instance takes as value for the attribute an instance of C', or of a subclass of C'. Moreover, an *aggregation* relationship is established between the two classes. An aggregation relationship between the class C and the class C' specifies that C is defined in terms of C'. Since C' can in turn be defined in terms of other classes, the set of classes in the schema is organized into an aggregation hierarchy. Actually, it is not a hierarchy in a strict sense, since class definitions can be recursive.

Extent and Persistence Mechanisms. Besides being a template for defining objects, in some systems the class also denotes the collection of its instances—that is, the class has also the notion of *extent.* The extent of a class is the collection of all the instances generated from this class. This aspect is important since the class is the basis on which queries are formulated, because queries are meaningful only when they are applied to object collections. In systems in which classes do not have the extensional function, the extent of each class must be maintained by the applications through the use of constructors such as the set constructor. Different sets can contain instances of the same class. Queries are thus formulated against such sets, and not against classes. The automatic association of an extent to each class (like in the ORION system) has the advantage of a simplifying the management of classes and their instances. By contrast, systems (like O_2 and GemStone) in which classes define only specification and implementation of objects and queries are issued against collections managed by the applications provide a greater flexibility at the price of an increased complexity in managing class extents.

An important issue concerns the persistence of class instances—that is, by which modalities objects are made persistent (that is, inserted in the database) and are eventually deleted (that is, removed from the database). In relational databases, explicit statements (like `INSERT` and `DELETE` in SQL) are provided to insert and delete data from the database. In object-oriented databases, two different approaches can be adopted with respect to object persistence:

- Persistence is an implicit property of all the class instances; the creation (through the new operation) of an instance has also the effect of inserting the instance in the database; thus the creation of an instance automatically implies its persistence. This approach is usually adopted in systems in which classes also have an extensional function. Some systems provide two different new operations: one for creating persistent objects of a class, the other one for creating temporary (transient) objects of that class.
- Persistence is an orthogonal properties of objects; the creation of an instance does not have the effect of inserting the instance in the database. Rather, if an instance has to survive the program that created it, it must be explicitly made persistent, for example, by assigning it a name or by inserting it in a persistent collection of objects. In some systems, an object is persistent if it is reachable from some persistent object. This approach is usually adopted in systems in which classes do not have the extensional function.

With respect to object deletion, two different approaches are possible:

- The system provides an explicit delete operation. The possibility of explicitly deleting objects poses the problem of referential integrity, if an object is deleted and there are other objects referring to it, references are no longer valid (such references are called dangling references). The explicit deletion approach is adopted by the ORION and Iris systems.
- The system does not provide an explicit delete operation. A persistent object is deleted only if all references to it have been removed (a periodic garbage collection is performed). This approach, adopted by the GemStone and O_2 systems, ensures referential integrity.

Migration. Because objects represent real-world entities, they must be able to reflect the evolution in time of those entities. A typical example is that of a person which is first of all a student, then an employee, then a retired employee. This situation can be modeled only if an object can become an instance of a class different from the one from which it has been created. This evolution, known as object migration, allows an object to modify its features—that is, attributes and operations—but still retaining its identity. Object migration among classes introduces, however, semantic integrity problems. If the value for an attribute A of an object O is another object O' (an instance of the class domain of A) and O' changes class and if the new class of O' is no longer compatible with the class domain of A, the migration of O' will result in O containing an illegal value for A. For this reason, migration is not currently supported in most existing systems.

Inheritance

Inheritance allows a class, called a subclass, to be defined starting from the definition of another class, called superclass. The subclass inherits attributes, operations, and methods of its superclass; a subclass may in addition have some specific, noninherited features. Inheritance is a powerful reuse mechanism. By using such a mechanism, when defining two classes, their common properties, if any, can be identified and factorized in a common superclass of theirs. The definitions of the two classes will, by contrast, specify only the distinguishing specific properties of these classes. This approach not only reduces the amount of code to be written, but it also has the advantage of giving a more precise, concise, and rich description of the world being represented.

Some systems allow a class to have several direct superclasses, in this case we talk of multiple inheritance, whereas other systems impose the restriction to a single superclass, in this case we talk of single inheritance. The possibility of defining a class starting from several superclasses simplifies the task of class definition. However, conflicts may arise. Such conflicts may be solved according to different strategies:

- An ordering is imposed on the superclasses, and conflicting features are inherited from the superclass preceding the others in the ordering;
- An explicit qualification mechanism is provided whereby the user explicitly specifies from which superclass each conflicting feature has to be inherited.

In scientific literature and in various object-oriented languages there are different inheritance notions. In the knowledge representation context, for instance, inheritance has quite a different meaning from the one it has in object-oriented programming languages. In the former context, a subclass defines a specialization with respect to features and behaviors of the superclass, whereas in the latter the emphasis is on attribute and method reuse. Different inheritance notions can then be considered, corresponding to which three different hierarchies can be distinguished:

- *Subtype Hierarchy:* expresses the consistency among type specifications by specifying subtype relationships supporting the substitutability of a subtype instance in each context where a supertype instance is expected (13);
- *Implementation Hierarchy:* supports code sharing among classes;
- *Classification Hierarchy:* expresses inclusion relationships among object collections.

Each hierarchy refers to different properties of the type/class system; those hierarchies are, however, generally merged in a single inheritance mechanism.

Overriding, Overloading, and Late Binding. The notion of overloading is related to the notion of inheritance. In many cases it is very useful to adopt the same name for different operations, and this possibility is extremely useful in the object-oriented context. Consider as an example (14) a `display` operation receiving as input an object and displaying it. Depending on the object type, different display mechanism are exploited: If the object is a figure, it should appear on the screen; if the object is a person, its data should be printed in some way; if the object is a graph, a graphical representation of it should be produced. Another problem arises for displaying a set of objects, the type of whose members is not known at compile-time.

In an application developed in a conventional system, three different operations `display_graph`, `display_person` and `display_figure` would be defined. This requires the programmer to be aware of all possible object types and all the associated display operations and to use them properly. Under a conventional approach, the application code performing the display of a set of objects on the screen would be organized as follows:

```
for x in X do
    begin
     case of type(x)
       person: display_person(x);
       figure: display_figure(x);
       graph: display_graph(x);
     end;
    end;
```

In an object-oriented system, by contrast, the `display` operation can be defined in a more general class in the class hierarchy. Thus, the operation has a simple name and can be used indifferently on various objects. The operation implementation is redefined for each class; this redefinition is known as overriding. As a result, a single name denotes different programs and the system takes care of selecting the appropriate one at each time during execution. Thus the code shown above is compacted as

```
for x in X do display(x)
```

This approach to design application code provides several advantages. The application programmers implementing the classes write the same number of methods, but the application designers do not have to take care of that. The resulting code is simpler and easier to maintain, since the introduction of a new class does not require us to modify the applications. At any moment, objects of other classes—for example, information on some products—can be added to the application and displayed by simply defining a class—for example, `product`—providing a proper (re)definition of the `display` operation. The important advantage is that the above compact application code would not require any modification. By contrast, the traditional application code would require modifications to deal with the new object classes.

To support this functionality, however, the system is no longer able to bind operation names to corresponding code at compile time; rather it must perform such binding at runtime: This late translation is known as late binding.

Thus, the notion of overriding refers to the possibility that a class will redefine attributes and methods it inherits from its superclasses; the inheritance mechanism allows thus to specialize a class through additions and substitutions. Overriding implies overloading, since an operation shared along a class hierarchy can have different implementations in the classes belonging to this class hierarchy; therefore, the same operation name denotes different implementations.

An Example

Figure 3 illustrates an example of object-oriented database schema. In the figure, each node represents a class. Each

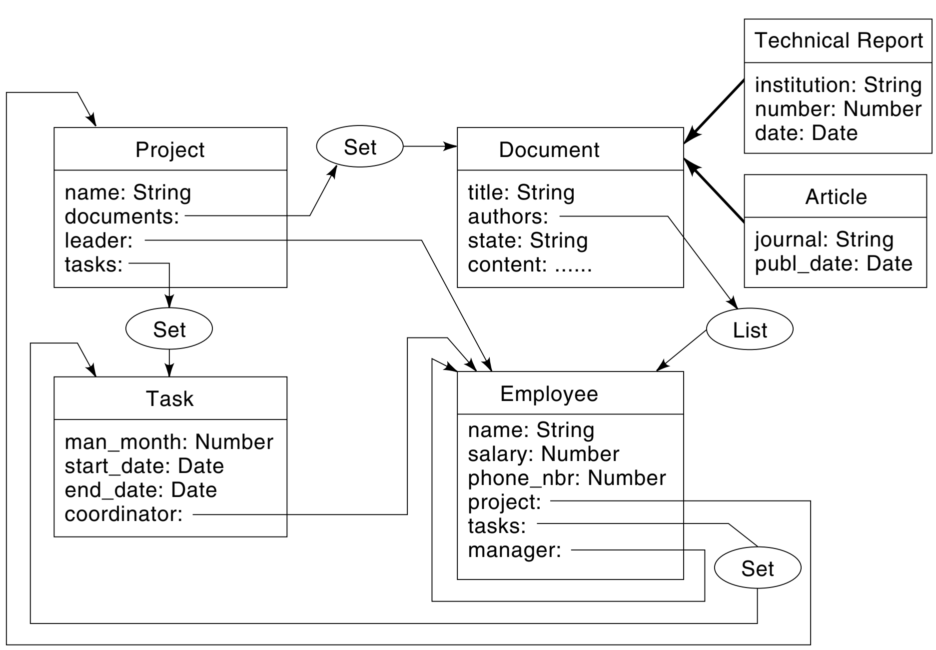

Figure 3. An example of object-oriented database schema that will be used in the text as a running example to discuss various systems.

node contains names and domains of the attributes of the class it represents. For the sake of simplicity, we have not included in the figure either operations or class features. Nodes can be connected by two different kinds of arcs. The node representing a class C can be linked to the node representing class C' through:

1. a thin arc, denoting that C' is the domain of an attribute A of C;
2. a bold arc, denoting that C is superclass of C'.

Note that the figure represents both the aggregation (thin arcs) and the inheritance (bold arcs) links among classes.

QUERY LANGUAGES

Query languages are an important functionality of any DBMS. A query language allows users to retrieve data by simply specifying some conditions on the content of those data. In relational DBMS, query languages are the only way to access data, whereas OODBMS usually provide two different modalities to access data. The first one is called navigational and is based on object identifiers and on the aggregation hierarchies into which objects are organized. Given a certain OID, the system is able to directly and efficiently access the object referred by it and can navigate through objects referred by the components of this object. The second access modality is called associative and is based on SQL-like query languages. These two different access modalities are used in a complementary way: A query is evaluated to select a set of objects which are then accessed and manipulated by applications through the navigational mechanism. Navigational access is crucial in many applications—like, for example, graph traversal. This type of access is inefficient in relational systems because it requires the execution of a large number of join operations. Associative access, by contrast, has the advantage of supporting the expression of declarative queries, thus reducing application development time. Most of the success of relational DBMS is because of their declarative query languages.

In the remainder of this section we point out the peculiar aspects of object-oriented query languages, emphasizing the features related to the new data model. We do not refer to any specific language. In the following section, we will present the GemStone and ObjectStore query languages and we will discuss OQL, the ODMG query language. For an extensive discussion on object-oriented query languages we refer the interested reader to Ref 15.

A first feature of object-oriented query languages is the possibility they offer of imposing conditions on nested attributes of an object aggregation hierarchy, through path expression, allowing us to express joins to retrieve the values of the attributes of an object components. In object-oriented query languages, therefore, two different kinds of join can be distinguished: implicit join, deriving from the hierarchical structure of objects, and explicit join, which, as in relational query languages, explicitly compares two objects. Other important aspects are related to the inheritance hierarchy and methods. First of all, a query can be issued against a class or against a class and all its subclasses. Most existing languages support both these possibilities. Methods can be used as derived attributes or as predicate methods. A method used as derived attribute is similar to an attribute; however, whereas the attribute stores a value, the method computes a value starting from data values stored in the database. A predicate method is similar, but it returns the boolean constants true or false. A predicate method evaluates some conditions on objects and can thus be part of the boolean expressions determining which objects satisfy the query.

Moreover, object-oriented query languages often provide constructs for expressing recursive queries, though recursion is not a peculiar feature of the object-oriented paradigm and it has already been proposed for the relational data model. It is, however, important that some kind of recursion can be expressed, since objects relevant for many applications are naturally modeled through recursion.

The equality notion also influences query semantics. The adopted equality notion determines the semantics and the execution strategy of operations such as union, difference, intersection, and duplicate elimination. Finally, note that external names that some object-oriented data models allow to associate with objects provide some semantically meaningful handlers that can be used in queries.

A relevant issue for object-oriented query languages is related to the language closure. One of the most remarkable characteristics of relational query languages is that the results of a query are, in turn, relations. Queries can then be composed; that is, the result of a query can be used as an operand in another query. Ensuring the closure property in object-oriented query language is, by contrast, more difficult. The main difficulty derives from the fact that often the result of a query is a set of objects whose class does not exist in the database schema and which is defined by the query. The definition of a new class "on-the-fly" as result of a query poses many difficulties, including where to position the new class in the inheritance hierarchies and which methods should be defined for such class. Moreover, the issue of generating OIDs for the new objects, namely, results of the query and instances of the new class, must be addressed.

To ensure the closure property, an approach is to impose restrictions on the projections that can be executed on classes. A restriction that is common to many query languages is that either all the object attributes or only a single attribute are returned by the query. Moreover, no explicit joins are supported by those languages. In this way the result of a query is always a set of already existing objects, instances of an already existing class; the class can be a primitive class (such as the class of integers, string, and so forth) or a user-defined class. If one wants to support more general queries with arbitrary projections and explicit joins, a first approach to ensure closure is to consider the results of a query as instances of a general class, accepting all objects and whose methods only allow to print or display objects. This solution, however, does not allow objects to be reused for other manipulations and therefore it limits the nesting of queries, which is the main motivation for ensuring the closure property.

Another possible approach is to consider the result of a query as a collection of objects, instances of a new class, which is generated by the execution of the query. The class implicitly defined by the query has no methods; however, methods for reading and writing attributes are supposed to be available as system methods. The result of a query is thus quite similar to a set of tuples. An alternative solution (12) is, finally, that of including relations in the data model and of defining the result of a query as a relation.

OBJECT-ORIENTED DBMSs

During recent years, several object-oriented database systems have been developed, both as experimental prototypes and as commercial systems. Among them, we recall the following: the ORION/Itasca system, developed at MCC; the Iris/OpenODB system, developed at Hewlett-Packard laboratories; the Ode system, developed at AT&T Bell Labs; the GemStone system of ServioLogic; the ObjectStore system of Object Design; the O_2 system of O_2 Technology; ONTOS of Ontologic; Objectivity of Objectivity Inc.; and Versant of Versant Technology. Those systems represent only a partial list of available OODBMS. Table 1 compares some of these systems along a number of dimensions.

In our comparison, we distinguish systems in which classes have an extensional function—that is, in which class the set of its instances is automatically associated—from those in which object collections are defined and handled by the applications. We point out, moreover, the adopted persistence mechanism, distinguishing among systems in which all objects are automatically created as persistent, systems in which persistence is ensured by linking an object to a persistence root (usually an external name), and systems supporting two different creation operations: one for creating temporary objects, the other one for creating persistent objects. The different policies with respect to encapsulation are also shown, distinguishing among systems forcing strict encapsulation, systems supporting direct accesses to attribute values, and systems distinguishing between private and public features.

An important concept which exists in many semantic models and in models for the conceptual design of databases (11) is the relationship. An relationship is a link between entities in applications. A relationship between a person and his employer (*) is one example; another (classic) example is the relationship between a product, a customer, and a supplier (**), which indicates that a given product is supplied to a given customer by a given supplier. Associations are characterized by a degree, which indicates the number of entities participating in the relationship, and by some cardinality constraints which indicate the minimum and maximum number of relationships in which an entity can participate. For example, relationship (*) has degree 2—that is, it is binary—and its cardinality constraints are (0,1) for person and (1,n) for employer. This reflects the fact that a person can have at most one employer, whereas an employer can have more than one employee. Referring to maximum cardinality constraint, relationships are partitioned in one-to-one, one-to-many, and many-to-many relationships. Finally, relationships can have their own attributes; for example, relationship (**) can have attributes `quantity` and `unit price`, indicating, respectively, the quantity of the product supplied and the unit price quoted. In most object-oriented data models, relationships are represented through object references. This approach, however, imposes a directionality on the relationship. Some models, by contrast, allow the specification of binary relationships without proper attributes.

Finally, the O_2 system allows the specification of exceptional instances—that is, of objects that can have additional features and/or redefine (under certain compatibility restrictions) features of the class of which they are instances.

In the remainder of this section we illustrate two specific systems, namely GemStone and ObjectStore.

GemStone

GemStone is an object-oriented database management system integrating the object-oriented programming language Smalltalk with the functionalities typical of a DBMS. The data definition and manipulation language is called OPAL and is a Smalltalk extension (16). As in Smalltalk, each system entity is considered an object, including OPAL programs. GemStone does not distinguish between objects and values; rather everything that is manipulated by the system is seen as an object.

Table 1. Comparison Among Data Models of Most Common OODBMSs

	GemStone	Iris	O_2	Orion	ObjectStore	Ode	ODMG
Reference:	31	32	33, 34	35, 36	37	38	39
Class extent:	No	Yes	No	Yes	No	Yes	Yes[a]
Persistence:	R	A	R	A	R	2op	A[b]
Explicit deletion:	No	Yes	No	Yes	Yes	Yes	Yes[b]
Direct access to attributes:	No	Yes	P	Yes	P	P	Yes
Domain specification for attributes	O	M	M	M	M	M	M
Class attributes and methods:	Yes	No	No	Yes	No	No	No
Relationships:	No	Yes	No	No	Yes	No	Yes
Composite objects:	No	No	No	Yes	No	No	No
Referential integrity:	Yes	No	Yes	No	Yes[c]	No	Yes[c]
Multiple inheritance:	No	Yes	Yes	Yes	Yes	Yes	Yes
Migration:	L	Yes	No	No	No	No	No
Exceptional instances:	No	No	Yes	No	No	No	No

R, root persistence; A, automatic; 2op, two different *new* operations; P, only for public attributes; O, optional; M, mandatory; L, in limited form.

[a] For those classes in which definition an `extent` clause is specified.

[b] In C++, OML created objects are automatically persistent and explicit deletion is supported; in Smalltalk, OML persistence is by root and there is no explicit delete operation.

[c] Referential integrity is ensured for relationships but not for attributes.

In GemStone methods and structures common to all the instances of a class are factorized in an object, referred to as CDO (class-defining object); thus a class itself is an object. All the instances of a class contain a reference to their CDO as part of their object identifier. Objects are characterized by their attributes, instance variables in GemStone terminology, whose values are references to other objects. The specification of attribute domains is not mandatory. Objects can be internally organized in complex structures, obtained by combining four different storage formats starting from atomic objects like integers and strings.

Data Definition in GemStone. A peculiar feature of GemStone is that it provides a hierarchy of predefined classes, called *kernel classes.* Each of those classes provides the structure and methods of most common data types, such as strings, booleans, arrays, sets, and so on. This class hierarchy imposes some criteria on attribute and method inheritance. The `Object` class is the root of that hierarchy; thus each class is subclass of `Object`. When a new class is defined, it must be defined as a subclass of an already existing class: either of the `Object` class or of one of the `Object` subclasses.

The syntax of the OPAL class definition statement is the following:

Superclass Name `subclass` *'Class Name'*
`instVarNames`: *Attribute List*
`classVars`: *Class Attribute List*
`poolDictionaries`: *Shared Attribute List*
`inDictionary`: *Dictionary Name*
`constraints`: *Domain Constraint List*
`instancesInvariant`: {`true` | `false`}
`isModifiable`: {`true` | `false`}

A subclass is defined by sending to the appropriate superclass (denoted in the above statement by *Superclass Name*) the `subclass` message, for which a method is specified in each class. Note that a class can have only a direct superclass; that is, GemStone does not support multiple inheritance. When a class receives a `subclass` message, it executes a method for the creation of a subclass named *Class Name,* whose characteristics are specified by other clauses in the class definition statement. In particular:

- The `instVarNames` clause takes as argument a list of string with format `#('string1','string2', . . .)`; each string specifies an attribute name.
- The `classVars` clause has as argument a list of class attribute names; recall that class attributes are attributes whose value is associated with the class rather than with its instances.
- The `poolDictionaries` clause takes as argument a list of shared attribute names; a shared attribute (pool variable) is a particular storage structure allowing different classes and their instances to share information.
- The `inDictionary` clause takes as argument the name of a predefined dictionary in which the name of the class being created is inserted; in such a way the class can be simply referred through its name.
- The `constraints` clause specifies attribute domains; note that in GemStone, domain specification is not mandatory; the name of that clause is due to the fact that in GemStone, domain specifications are seen as an integrity constraint specification.
- The `instancesInvariant` clause specifies whether or not the class instances can be modified; the clause argument is `true` if no modifications are allowed, while it is `false` otherwise; if the clause has as argument `true` the objects, instances of the class, can be modified only during the transaction that created them, after the end of that transaction they can no longer be modified.
- The `isModifiable` clause specifies whether or not the class can be modified; modifications to a class include addition and deletion of attributes.

Classes whose `isModifiable` clause has `true` value cannot be instantiated. Therefore, it is not possible to modify the schema of classes that have already been populated, since this would require a modification of all the class instances.

In GemStone it is not possible to define a class in terms of classes which have not yet been defined; thus database schema whose aggregation hierarchies contain cycles cannot be defined directly. A class definition can, however, be modified after having been defined, by addition of some domain constraints for its attributes. The class must then be initially declared as modifiable; that is, the `isModifiable` clause must contain the `true` value. Once the class which is the attribute domain has been defined, the first class definition can be modified through the invocation of the message:

Class Name `instVar`: *'Attribute Name'* `constrainTo`: *Domain.*

This message takes two arguments: The first one is introduced by the keyword `instVar:` and denotes an attribute; the second one is introduced by the keyword `constrainTo:` and denotes a class. The effect of this message is to add a domain constraint to the class receiver of the message. At this point, the class can be made nonmodifiable through the operation `immediateInvariant` provided by the system. Once the class has been made nonmodifiable, it can be instantiated.

A possible OPAL definition for the database schema from Fig. 3 is the following, in which some class definitions are omitted for the sake of brevity.

```
Object subclass 'Employee'
       instVarClassNames: #('name','salary',
                            'phone_nbr',
                            'manager',
                            'project','tasks')
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[#[#name,String],
                      #[#salary,Integer],
                      #[phone_nbr,Integer]]
       instancesInvariant: false
       isModifiable: true.
Set subclass 'Employees'
       instVarClassNames: #()
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: Employee
       instancesInvariant: false
       isModifiable: false.
```

```
Object subclass 'Document'
       instVarClassNames: #('title','authors',
                            'state','content')
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[#[#title,String],
                      #[#authors,Employees],
                      #[#state,String],
                      #[#content,String]]
       instancesInvariant: false
       isModifiable: false.
Set subclass 'Documents'
       ...
Document subclass 'Article'
       instVarClassNames: #('journal',
                            'publ_date')
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[#[#journal,String],
                      #[#publ_date,Date]]
       instancesInvariant: false
       isModifiable: false.
Documents subclass 'Articles'
       ...
Document subclass 'Technical_Report'
       ...
Documents subclass 'Technical_Reports'
       ...
Object subclass 'Task'
       ...
Set subclass 'Tasks'
       ...
Object subclass 'Project'
       instVarClassNames: #('name','documents',
                            'tasks','leader')
       classVars: #()
       poolDictionaries: #()
       inDictionary: UserGlobals
       constraints: #[#[#name,String],
                      #[#documents,Documents],
                      #[#tasks,Tasks],
                      #[#leader,Employee]]
       instancesInvariant: false
       isModifiable: false.
Set subclass 'Projects'
       ...
Employee instVar: 'manager'
         constrainTo:Employee.
Employee instVar: 'project'
         constrainTo:Project.
Employee instVar: 'tasks' constrainTo:Tasks.
Employee immediateInvariant.
```

Note that each class definition is followed by the definition of a subclass of the kernel class `Set` whose `constraints` class has the first class as argument. For instance, the definition of class `Employee` is followed by the definition of the class `Employees`, whose instances are sets of objects belonging to class `Employee`. In such a way the extent of the class is specified, since in GemStone classes have no extensional functionality. Each object collection in OPAL is defined as a (either direct or indirect) sublcass of the kernel class `Set`. Usual operations on sets are inherited from the `Set` class. A collection can then be used as attribute domain in other classes.

In OPAL a method specification consists of two components: a *message pattern,* representing the method signature, and an implementation. As in Smalltalk, a message in OPAL denotes the invocation of an operation on an object. A message consists of:

- an OID or an expression, denoting the object to which the message is sent;
- one or more identifiers, referred to as selectors, specifying the method to be invoked;
- possibly one or more arguments; arguments can in turn be denoted by message expressions.

Messages are classified in three categories:

1. *Unary Messages.* The simplest kind of message consists of a receiver object and a single selector. An example is the expression `7 negated`, returning `−7`.
2. *Binary Messages.* A binary message consists of a receiver, a selector, and a single argument. An example is the expression `myObject = yourObject` returning the boolean constant *true* if the two objects `myObject` and `yourObject` have the same value, whereas the expression `myObject == yourObject` returns *true* if the two objects are identical, that is, if they have the same OID.
3. *Keyed Messages.* A keyed message consists of a receiver and several key-argument pairs (up to a maximum of 15). Each key is a simple identifier terminated by the character ':'. An example is the message expression `Employee instVar: 'tasks' constrainTo:Tasks`, which contains two key-argument pairs; the first pair has `instVar:` as key and an attribute name as argument, the second one has `constrainTo:` as key and a class name (the domain for the attribute) as argument.

Messages as the ones illustrated above can be combined, and messages can be as well sent in cascade to the same object.

A method implementation consists of:

- declarations of temporary variables;
- one or more OPAL expressions; this language includes expressions typical of programming languages such as conditional expressions and assignments;
- a return statement which returns a value for the message expression which has invoked the method.

Note that in GemStone, object attributes are directly accessible only by the object methods. Thus, to simply read or modify an attribute the appropriate methods must be defined. The following example illustrates how methods for reading and modifying values of the `title` attribute of the `Document` class can be defined.

Consider the following OPAL method definition statements:

```
method: Document
title
        ^title
%
method: Document
title:  aTitle
        title := aTitle
%
```

In the above definitions, the clause `method: Document` denotes that a method of the `Document` class is being defined. The character `%` is a statement terminator. The character `^` denotes the method return value. The above methods have the same name, that is, `title`, but different signatures: The first method is indeed an unary message, while the second one is a keyed message. The system determines depending on the message type which code to use to answer an invocation. For example, for the invocation `aDocument title` the first method will be executed, whereas for the invocation `aDocument title: 'Databases'` the second one will be executed.

GemStone allows the definition of class methods. The following OPAL method definition defines a method for creating and initializing objects of the `Document` class.

```
classmethod: Document
  nTitle: aTitle      nAuthors: AuthorSet
  nState: aState      nContent: aContent
  |tempDoc|
  tempDoc := self.new.
  tempDoc title: aTitle; authors: AuthorSet;
          state: aState; content: aContent;
  ^tempDoc
%
```

The method creates a new instance of the `Document` class and it assigns the method arguments to the instance attributes. The method also contains the declaration of a temporary variable `|tempDoc|`. The first statement creates an instance of the `Document` class through the `new` operation and assigns it to the temporary variable. Then, four messages are sent in cascade to that instance for initializing its attributes.

As we have already said, GemStone supports single inheritance. When defining a subclass, new attributes and methods can be added, and methods can be redefined. New attributes can always be added; the only exception is when the superclass instances have the set or collection as structure—for example, the `Documents` class above. Class attributes and shared attributes can be added as well. If the instance attributes of a class have domain constraints, the constraints are inherited by the subclasses. An inherited domain constraint can be modified, but it can only be made more restrictive; that is, the domain specified in the subclass must be a subclass of the domain specified in the superclass. A method can be redefined by defining in the subclass a method with the same message schema and a different implementation. A method can also be redefined. Method refinement is the addition of code to an inherited method implementation. In OPAL, as in Smalltalk, the pseudovariable `super` can be used to refer to the superclass method which is being refined.

Data Manipulation in GemStone. With respect to persistence, GemStone falls in the category of systems in which objects are not automatically persistent. The simplest way to make an object persistent is to associate an external name with the object. The statement for assigning a name to an object is the following:

Dictionary Name `at:` *Name* `put:` *Object*

Each object which can be reached by a persistent object is itself persistent. A common approach is to define a set of instances and to make this set persistent (for instance, by assigning it a name). All the objects belonging to this set are therefore made persistent, even if they have no explicitly associated external names. The following statement sequence define a persistent collection of projects `myProjects` and inserts into it a newly created project.

```
|Proj aProj|
Proj := Projects new.
aProj := Project new.
Proj add: aProject.
UserGlobal at: #myProjects put: Proj.
```

To delete an object, all its persistence roots must be removed. Then, the object is automatically deleted. GemStone does not provide an explicit delete operation; rather it relies on a *garbage collection* mechanism. Referring to the project object above, it can be deleted by simply removing it from the persistent collection through the statement

```
myProjects remove: aProject
```

while for deleting the collection including all projects the association between the object and its name must be removed, through the statement

```
UserGlobal at: #myProjects put: nil.
```

GemStone supports a limited form of object migration. The message for requiring the migration of an object is `changeClassTo`, whose only argument is the class into which the object migrates. An object can only migrate from a class to a subclass, which cannot have redefined attribute domains and whose storage format must be the same as that of the class from which the object migrates. Moreover, the subclass cannot have additional instance attributes.

Queries in GemStone can be issued only against objects defined as set, whose instances must belong to a class whose attribute domains are specified. Queries are expressed through a special `select` message. This message takes as argument a string denoting a combination of boolean predicates defining the query. The query result is a set of objects whose class is the same as the one of the objects against which the query is issued. Queries can also contain path expressions. The following OPAL query returns all tasks with a manpower greater than 20 months whose coordinator earns more than 20000:

```
Tasks select: {:t | (t.man_month > 20 &
                     t.coordinator.salary > 20000) }
```

In addition to the `select` message, the query language supports other query protocols. In particular, the `reject` message selects all the objects that do not satisfy a given predicate, while the `detect` message returns an object satisfying the predicate.

ObjectStore

The ObjectStore system is tightly integrated with the C++ language, and it provides the possibility of storing in persistent memory C++ objects. This approach allows us to overcome the impedance mismatch problem (17) between a programming language and a DBMS, where the data structures provided by the DBMS are different from those provided by the programming language. ObjectStore can exploit the C++ class definition language as data definition language. Actually, an extended C++ including additional constructs for data handling is used. Objects in a class can be transient; that is, they can be deleted at the end of program execution or can be persistent—that is, permanently stored in the database. Objects, however, are manipulated according to the same modalities independently from their persistence status. Persistence is thus an orthogonal property with respect to object use. Persistent objects can be shared by several programs.

Data Definition in ObjectStore. C++ distinguishes between objects and values, and so does ObjectStore. In particular, instances of base types character (`char`), integer (`int`), real (`float`), and string (`char*`) are values; moreover, the `struct` constructor allows us to specify structured values. An asterisk (*) is used to specify a reference (pointer). ObjectStore extends C++ with a set constructor. Set types can be specified by declarations of the form `os_Set` <*Argument Type*>, where *Argument Type* is the type of the objects in the set being defined. For example, the type `os_Set <Document*>` is a set of pointers to objects of type `Document`. ObjectStore also extends C++ with multiset (bag) and list constructors; those constructors are `os_Bag` and `os_List`, respectively.

The syntax of ObjectStore class definition statement is the following:

```
class Class Name: superclass_spec {
        public:  Public Attribute List
                 Public Method List
        private: Private Attribute List
                 Private Method List
}
```

In the above statement:

- `superclass_spec` is a list of superclasses, specified as `public` *Superclass Name* or as `private` *Superclass Name;* if no specification for the inheritance modality is specified for a superclass, that is, the superclass name is preceded neither by `private` nor by `public`, the class inherits in a private way. The difference between inheriting from a class in a public or private way is related to attribute and method visibility, and it is the same as in C++. In particular, the private features of the superclass are not visible in the subclass in both cases, whereas public features of the superclass are (a) public properties of the subclass, if the subclass inherits in a public way and (b) private properties of the subclass, if the subclass inherits in a private way. In what follows, for the sake of simplicity, we will restrict ourselves to consider subclasses inheriting in a public way.
- The `public` clause introduces the list of declarations of public features (attributes and methods) of the class; these features can be directly accessed from outside the objects. The `private` clause, by contrast, introduces the list of declarations of private features (attributes and methods) of the class; these features can be accessed only within methods of the class. Actually, in the class definition some features can be specified before the `public:` keyword; all the properties specified before this keyword are private, that is, they are visible only within the class.
- Each attribute in the list is declared as

  Domain AttributeName;

  where *Domain* is either a base type, a structured type, or a class name or is a pointer to one of these.
- Each method signature in the list is declared as

  Return Value Type Method Name (*Arguments*);

  Methods are distinguished from attributes in that after the name specification they contain the specification of parameters enclosed within brackets; if a method has no parameter, only the brackets () are included. A method returning no value has type `void`.

The ObjectStore definition of classes `Employee, Document, Article` and `Project` of the database schema of Fig. 3 is the following:

```
struct Date {
   int   month;
   int   day;
   int   year;
}

class Employee {
    public:
      char*          name;
      int            salary;
      int            phone_nbr;
      Employee*      manager;
      Project*       project;
      os_Set<Task*> tasks;
      int bonus();
}

class Document {
    public:
      char*               title;
      os_List<Employee*> authors;
      char*               state;
      char*               content;
}

class Article: public Document {
    public:
      char* journal;
      Date  publ_date;
}
```

```
class Project {
    public:
      char*              name;
      os_Set<Document*> documents;
      os_Set<Task*>     tasks;
      Employee*          leader;
}
```

A further extension of ObjectStore to C++ is related to the notion of *relationship*. This extension allows the specification of inverse attributes, representing binary relationships. This functionality is requested through the keyword `inverse_member` associated with an attribute and followed by the inverse attribute name. ObjectStore automatically ensures relationship consistency. As an example, the relationship between an employee and a project corresponding to the fact that the employee leads the project can be modeled by the inverse attributes `leads` in `Employee` and `leader` in `Project`. The ObjectStore class declarations are as follows:

```
class Employee {
    ...
    Project* leads
       inverse_member Project::leader;
    ... }
class Project {
    ...
    Employee* leader
       inverse_member Employee::leads;
    ... }
```

Through the `os_Set` constructor, one-to-many and many-to-many relationships can be represented as well. Consider for example the relationship between an employee and a task, corresponding to the fact that the employee participates in the task. This relationship can be modeled by the inverse attributes `tasks` in `Employee` and `members` in `Task`. The ObjectStore class declarations are as follows:

```
class Employee {
    ...
    os_Set<Task*> tasks
       inverse_member Task::members;
    ... }
class Task {
    ...
    os_Set<Employee*> members
       inverse_member Employee::tasks;
    ... }
```

In ObjectStore, method implementation is specified through the C++ language extended with methods defined for the collection types `os_Set`, `os_Bag` and `os_List`. Those methods include `insert(e)`, `remove(e)` and `create` which, respectively, insert and delete an object from a collection and create a new collection. A `foreach (e,c)` statement for iterating over the element `e` of a collection `c` is also provided.

As an example, consider the following methods of class `Task`:

```
void change_coord(Employee* ncoord);
void delete_part(Employee* part);
void add_part(Employee* part);
int salary_budget();
```

These operations change the task coordinator, delete and add a participant to the task, and compute the sum of the salaries of employees assigned to the task, respectively. The following are possible implementations for those operations:

```
void Task::change_coord(Employee* ncoord)
   { coordinator = ncoord; }
void Task::delete_part(Employee* part)
   { participants -> remove(part); }
void Task::add_part(Employee* part)
   { participants -> insert(part); }
int Task::salary_budget()
   { int sum = 0; Employee* e;
     foreach(i,participants) {
        sum += e -> salary;
     }
     return sum;
   }
```

Another C++ feature inherited by ObjectStore is related to class *constructors*. A class can have a method whose name is the same of the class name; this method is executed each time a new object of the class is created. Constructors can have parameters; several constructors can also be associated with the same class (obviously, the number of parameters must be different).

In ObjectStore, as in GemStone, inherited methods can be redefined.

Data Manipulation in ObjectStore. In ObjectStore, as in GemStone, persistence is not an automatic property of objects. To create an object or a persistent collection of objects in ObjectStore the application must assign it a *name*, which is also referred to as *persistent variable*. This name can be seen as a persistent reference, stored by the system, to the object. The statement for assigning a name to an object at object creation time has the following format:

$$Type\,\&Name = Type{::}\,\texttt{create}(DB\_Name);$$

An object belonging to a persistent collection of objects is automatically made persistent.

The following ObjectStore statements illustrate the specification of a collection `Employees`, and the creation of an object belonging to the class `Employee` which is made persistent by inserting it in the collection:

```
...
os_Set<Employee*> &Employees = os_Set<Employee*>
::create(my_db);
Employee* e = new(my_db) Employee;
Employees.insert(e);
...
```

ObjectStore, as C++, supports explicit object deletion, through the `delete` operation. Referential integrity is ensured for relationships but not for attributes. For what concerns relationships, upon the deletion of a participating object, the relationship is also deleted. Thus, no dangling references can arise. It can also be specified that the object participating in the relationship with the deleted object must in turn be deleted.

ObjectStore also provides a query language, which can be used to select a set of objects from a collection by specifying a selection condition. The query result is a set of pointers to objects satisfying the condition. The statements of the query language can be hosted in the C++ language.

The query returning all tasks with a man power greater than 20 months whose coordinator earns more than 20000 is expressed in ObjectStore as follows:

```
os_Set<Task*> &sel_tasks =
   Tasks [: man_month > 20 &&
            coordinator [: salary > 20000 :] :]
```

THE ODMG STANDARD

ODMG-93 is an OODBMS standard, consisting of a data model and a language, which has been proposed in 1993 by a consortium of major companies producing OODBMS (covering about 90% of the market). This consortium includes as voting members Object Design, Objectivity, O_2 Technology, and Versant Technology and includes as nonvoting members HP, ServioLogic, Itasca, and Texas Instruments. The ODMG-93 standard consists of the following components:

- an object data model (ODMG Object Model);
- an object data definition language (ODL);
- an object query language (OQL):
- interfaces for the object-oriented programming languages C++ and Smalltalk, and data manipulation languages for those languages (C++ OML and Smalltalk OML).

The ODMG Object Model is a superset of the OMG (Object Management Group) Object Model that gives it database capabilities, including relationships, extents, collection classes, and concurrency control. The Object Definition Language is a superset of OMG's Interface Description Language (IDL) component of CORBA (Common Object Request Broker Architecture), the emerging standard for distributed object-oriented computing developed by OMG.

Data Definition in ODMG

ODMG supports both the notion of object and the notion of value (literal in the ODMG terminology). Literals can belong to (a) atomic types such as long, short, float, double, boolean, char, and string, (b) types obtained through the set, bag, list, and array constructors, (c) enumeration types (`enum`), and (d) the structured types date, interval, time, and timestamp. Objects have a state and a behavior. The object state consists of a certain number of properties, which can be either attributes or relationships. An attribute is related to a class, while a relationship is defined between two classes. The ODMG model only supports binary relationships—that is, relationship between two classes: One-to-one, one-to-many, and many-to-many relationships are supported. A relationship is implicitly defined through the specification of a pair of traversal paths, enabling applications to use the logical connection between objects participating in the relationship. Traversal paths are declared in pairs, one for each traversal direction of the binary relationship. The `inverse` clause of the traversal path definition specifies that two traversal paths refer to the same relationship. The DBMS is responsible for ensuring value consistency and referential integrity for relationships. This means that, for example, if an object participating in a relationship is deleted, any traversal path leading to it is also deleted.

The ODMG class definition statement has the following format:

```
interface Class Name: Superclass List
[(extent Extent Name
  key[s] Attribute List ]
{ persistent | transient }
{

          Attribute List
          Relationship List
          Method List
}
```

In the above statement:

- the `extent` clause specifies that the extent of the class must be handled by the OODBMS;
- the `key[s]` clause, which can appear only if the `extent` clause is present, specifies a list of attributes for which two different objects belonging to the extent cannot have the same values;
- each attribute in the list is specified as

  `attribute` *Domain Name;*

- each relationship in the list is specified as

  `relationship` *Domain Name*
  `[inverse` *Class Inverse Name*`]`

  where *Domain* can be either *Class,* in the case of unary relationships, or a collection of *Class* elements, and *Inverse Name* is the name of the inverse traversal path, whose specification is optional;

- each method in the list is specified as

  Type *Name*(*Parameter List* `[raises` *Exception List*`]`

  where *Parameter List* is a list of parameters specified as

  `in | out | inout` *Parameter Name*

  and the `raises` clause allows to specify the exceptions that the method execution can raise.

The ODL definition of classes `Employee, Document, Article, Project` and `Task` of the database schema of Fig. 3, extended with the relationships between employees and projects, and between employees and tasks introduced above, is the following:

```
interface Employee
(   extent Employees
    key name) : persistent
{
    attribute string name;
    attribute unsigned short salary;
    attribute unsigned short phone_nbr[4];
    attribute Employee manager;

    relationship Project project;
    relationship Project leads
       inverse Project::leader;
    relationship Set<Task> tasks
       inverse Task::participants;
    int bonus();
}

interface Document
(   extent Documents
    key title) : persistent
{
    attribute string title;
    attribute List<Employee> authors;
    attribute string state;
    attribute string content;
}

interface Article: Document
(   extent Articles) : persistent
{
    attribute string journal;
    attribute data publ_date;
}

interface Project
(   extent Projects
    key name) : persistent
{
    attribute string name;
    attribute Set<Document> documents;
    attribute Set<Task> tasks;
    relationship Employee leader
       inverse Employee::leads;
}

interface Task
(   extent Tasks) : persistent
{
    attribute unsigned short man_month;
    attribute date start_date;
    attribute date end_date;
    attribute Employee coordinator;

    relationship Set<Employee> participants
       inverse Employee::tasks;
}
```

Note that, as in the above example, we have arbitrarily chosen some links between classes as object-valued attributes (for example, attribute `coordinator` in class `Task`) and some others as relationship for which a single traversal path, but not the inverse one, is specified (for example, traversal path `project` in class `Employee`). The main difference in representing a link between objects as a relationship rather than as a reference (that is, attribute value) is in the nondirectionality of the relationship. If, however, only one direction of the link is interesting, as in the two examples above, the link can indifferently be represented as an attribute or as a traversal path without inverse path. In this second case, however, the system ensures referential integrity, which is not ensured if the link is represented as an attribute.

ODMG does not specify any method definition language, since the idea is to allow using any object-oriented programming language (C++, Smalltalk, etc.).

Data Manipulation in ODMG

ODMG does not support a single DML, rather two different DMLs are provided, one related to C++ and the other one to Smalltalk. These OMLs are based on different persistence policies, corresponding to different object handling approaches in the two languages. For example, C++ OML supports an explicit delete operation (`delete_object`), while Smalltalk OML does not support explicit delete operations rather it is based on a garbage collection mechanism.

ODMG, by contrast, supports an SQL-like query language (OQL), based on queries of the `select-from-where` form. The query returning all tasks with a manpower greater than 20 months whose coordinator earns more than 20000, is expressed in OQL as follows:

```
select t
from Tasks t
where t.man_month > 20 and
      t.coordinator.salary > 20000
```

OQL is a functional language in which operators can be freely composed, as a consequence of the fact that query results have a type which belongs to the ODMG type system. Thus, queries can be nested. As a stand-alone language, OQL allows to query object denotable through their names. A name can denote an object of any type (atomic, collection, structure, literal). The query result is an object whose type is inferred from the operators in the query expression. The result of the query "retrieve the starting data of tasks with a manpower greater than 20 months," expressed in OQL as

```
select distinct t.start_date
from Tasks t
where t.man_month > 20
```

is a literal of type `Set<date>`.

The result of the query "retrieve the starting and ending dates of tasks with a manpower greater than 20 months," expressed in OQL as

```
select distinct struct(sd: t.start_date,
ed: t.end_date)
from Tasks t
where t.man_month > 20
```

is a literal of type `Set<struct(sd : date, ed : date)>`.

A query can return structured objects having objects as components, as it can combine attributes of different objects. Consider as an example the following queries. The query "retrieve the starting date and the coordinator of tasks with a man power greater than 20 months," expressed in OQL as

```
select distinct struct(st: t.start_date,
c: coordinator)
from Tasks t
where t.man_month > 20
```

produces as result a literal with type `Set<struct(st : date, c : Employee)>`. The query "retrieve the starting date, the names of the coordinator and of participants of tasks with a man power greater than 20 months," expressed in OQL as

```
select distinct struct(sd: t.start_date,
                       cn: coordinator.name,
                       pn: (select p.name
                            from t.participants as p))
where Tasks t
where t.man month > 20
```

produces as result a literal with type `Set<struct(st : date,cn : string, pn : bag<string>)>`.

OQL is a very rich query language. In particular it allows us to express, in addition to path expressions and projections on arbitrary sets of attributes, illustrated by the above examples, explicit joins and queries containing method invocations. The query "retrieve the technical reports having the same title of an article" is expressed in OQL as

```
select tr
from Technical_Reports tr, Articles a
where tr.title = a title
```

The query "retrieve the name and the bonus of employees having a salary greater than 20000 and a bonus greater than 5000", is expressed in OQL as

```
select distinct struct(n: e.name, b: e.bonus)
from Employees e
where e.salary > 20000 and e.bonus > 5000
```

OQL finally supports the aggregate functions `min`, `max`, `count`, `sum`, and `avg`. As an example, the query "retrieve the maximum salary of coordinators of tasks of the CAD project" can be expressed in OQL as

```
select mix(select e.salary
           from p.tasks.coordinator e)
from Projects p
where p.name = 'CAD'
```

OBJECT RELATIONAL DATABASES

As discussed at the beginning of this article, DBMSs are currently used by a large variety of applications. Each type of application is characterized by different requirements toward data handling. The most relevant application types include:

- business applications, which are characterized by large amounts of data, with a simple structure, on which more or less complex queries and updates are executed; the data must be accessed concurrently by several applications, and functionalities for data management (such as access control) are required;
- complex navigational applications, which include applications such as CAD and telecommunications; they need to manipulate data whose structures and relationships are complex and to efficiently traverse such relationships;
- multimedia applications, which require storage and retrieval of images, texts and spatial data, in addition to data representable in tables; they require the definition of application-specific operations, along with the integration of data and operations from different domains.

Currently, neither the relational DBMS nor the OODBMS fully meet all the requirements of all those application types:

- Relational DBMS handle and manipulate simple data; they support a query language (SQL) well-suited to model most business applications, and they offer good performance, multi-user support, and access control and reliability
- OODBMS allow us to directly represent complex objects and efficiently support navigational applications; however, they do not offer access control mechanisms and provide a limited support for concurrency and simple transactional models; moreover, though most of them provide declarative query languages, those languages are not thought of as an essential feature of an OODBMS.

We can thus say that relational DBMS provide an excellent support to applications manipulating simple data, whereas object-oriented DBMS provide an efficient support for applications manipulating complex data, but without some of the functions of relational DBMS, such as powerful declarative, high-level query languages, data security, concurrency control, and recovery. Object relational DBMS (18) have recently been proposed to overcome the shortcoming of relational DBMS and OODBMS. Object relational DBMS extend relational systems with the modeling capabilities of OODBMS, thus supporting complex operations on complex data. Object relational DBMS are motivated by the need of providing a rich data model, able to represent complex data as in the OODBMS, by supporting at the same time all the data management functions that relational DBMSs provide for the simple data they manage. Object relational DBMS include DB2 (19), UniSQL (20), Illustra/Informix (21), Oracle (22), Sybase (23). All these systems extend a relational DBMS with object-oriented modeling features. In all those DBMS the type system has been extended in some way, and the possibility of defining methods to model user-defined operations on types has been introduced. In what follows we briefly discuss the most relevant type system extensions.

Type System Extensions

Primitive Type Extensions. Most DBMS support predefined types such as integers, floating points, strings, and dates. Object relational DBMS support (a) the definition of new primitive types starting from predefined primitive types and (b) the definition of user-defined operations for these new primitive types. Operations on predefined types are inherited by the user-defined type, unless they are explicitly redefined. Consider as an example a `yen` type, corresponding to the Japanese currency. In a relational DBMS, this type is represented as a numeric type with a certain scale and precision—for example, DECIMAL(8,2). The predefined operations of the DECIMAL type can be used on values of this type, but no other operations are available. Thus, any additional semantics—for instance, converting yens to dollars—must be handled by the application, as the display in an appropriate format of values of that type.

In an object relational DBMS, by contrast, a type yen can be defined, and the proper functions can be associated with it, as illustrated by the following statements:

```
CREATE DISTINCT TYPE yen AS Decimal(8,2)
MEMBER FUNCTION add(yen,yen) RETURNS yen,
DISPLAY FUNCTION display(yen) RETURNS CHAR(11);
```

Complex Types. A complex, or structured, type includes one or more attributes. This notion corresponds to the notion of struct of the C language or to the notion of record of the Pascal language. Complex types are called *named row types* in SQL-3 (24). As an example, consider the type t_Address, defined as follows:

```
CREATE TYPE t_Address (street   VARCHAR(50),
                       number   INTEGER,
                       city     CHAR(20),
                       country  CHAR(2),
                       zip      INTEGER);
```

Relations can contain attributes whose type is a complex type. These relations are called *object tables* or *named row type tables* in SQL-3. For example, given the t_Address type defined above, the following is a definition of a named row type table:

```
CREATE TABLE EMPLOYEES (name       CHAR(20),
                        emp#       INTEGER,
                        curriculum TEXT,
                        address    t_Address,
                        dept       REF t_Department,
                        projects   TABLE OF REF
                                   t Project);
```

This relation can be equivalently defined as

```
CREATE TYPE t_Employee (name       CHAR(20),
                        emp#       INTEGER,
                        curriculum TEXT,
                        address    t_Address,
                        dept       REF t_Department,
                        projects   TABLE OF REF
                                   t_Project);
CREATE TABLE Employees OF t Employee;
```

Components of attributes, whose domain is a complex type, are accessed by means of the nested dot notation. For example, the zip code of the address of an employee is accessed as Employees.address.zip.

Methods can be defined on complex types, as part of the type definition. The definition of the type t_Employee can, for example, be extended with the definition of some methods as follows:

```
CREATE TYPE t_Employee (name       CHAR(20),
                        emp#       INTEGER,
                        curriculum TEXT,
                        address    t_Address,
                        dept       REF t_Department,
                        projects   TABLE OF REF
                                   t_Project);
MEMBER FUNCTION last_name(t_Employee)
   RETURNS CHAR(10),
MEMBER FUNCTION cmpare(t_Employee,t_Employee)
   RETURNS BOOLEAN;
```

With each complex type a constructor type, having the same name of the type, is associated. This method creates an instance of the type, given its attribute values. As an example, the invocation t_Address('Via Comelico', 39, 'Milano', 'I', 20135) creates a value of t_Address type. The application must, moreover, provide methods for comparing and ordering values of complex types.

Encapsulated Types. Encapsulated types are types whose content can be accessed only through methods. For example, if t_Address had been defined as an encapsulated type, its structure could only be accessed through methods. Those methods are called *accessors* and *mutators.* Thus, an accessor method should be defined for accessing the street attribute, another one should be defined for accessing the number attribute, and so on. These types are called value adts in SQL-3. The statement for defining an encapsulated type is CREATE VALUE TYPE instead of CREATE TYPE.

Reference Types. Reference types model the relationships among type instances. Those types allow a column in a relation to refer to a tuple in another relation. A tuple in a relation is identified through its OID. Given the declarations

```
CREATE TYPE t_Department (name       CHAR(10),
                          dept#      INTEGER,
                          chair      REF t_Employee,
                          dependents TABLE OF REF
                                     t_Employee,
                          map        PICTURE);
CREATE TABLE Departments of t Department;
```

and the above declarations of the type t_Employee and the relation Employees:

- The dept column of the Employees relation refers to a tuple of the Departments relation (corresponding to the department the employee works in).
- The chair column of the Departments relation refers to a tuple of the Employees relation (corresponding to the department chair).

A complex type cannot recursively contain a component of the same type; however, it can contain a reference to another object of the same type. To represent the manager of an employee, the t_Employee type could be extended to include a manager attribute defined as follows:

```
CREATE TYPE t_Employee ( ...
                        manager REF t_Employee,
                        ... );
```

The attributes of a referred instance can be accessed by means of the dot notation. For example, referring to the example above, the name of the department the employee works in is `Employees.dept.name`, while the name of a department chair is `Departments.chair.name`.

Collection Types. Object relational DBMS support constructors for grouping several instances of a given type. Those constructors model collections of type instances and include `SET, MULTISET, LIST, TABLE` (multiset of tuples). Referring to the `Departments` relation above, the `dependents` attribute is a collection of values of the `t_Employee` type. An attribute declared as

```
a_emp ARRAY OF REF t_Employee
```

represents by contrast an array of references to instances of the `t_Employee` type.

Elements of the collections are denoted by indexes in the case of arrays (for example, `a_emp[5]` denotes the fifth employee in the array), whereas multisets and tables can be iterated over through an SQL query as any other table. The SQL statement

```
SELECT d.name, (SELECT e.name
                FROM d.Employees e
                WHERE e.emp# > 1000)
FROM Department d
WHERE d.dept# = 777;
```

returns the department name and the names of a set of employees.

Inheritance. Inheritance specifies subtype/supertype relationships among types. Subtypes inherit attributes and methods of their supertypes. Object relational DBMS allow us to specify inheritance relationships both among types and among relations. The following declarations specify types `t_Student` and `t_Teacher` as subtypes of the `t_Person` type:

```
CREATE TYPE t_Person (name    CHAR(20),
                      ssn     INTEGER,
                      b_date  DATE,
                      address t_Address);
CREATE TYPE t_Teacher (salary  DECIMAL(8,2),
                       dept    REF t_Department,
                       teaches TABLE OF REF t_Course)
UNDER t_Person;
CREATE TYPE t_Student (avg_grade FLOAT,
                       attends  TABLE OF REF t_Course)
UNDER t Person;
```

The following declarations, by contrast, specify inheritance relationships among relations:

```
CREATE TABLE Persons OF t_Person;
CREATE TABLE Teachers OF t_Teacher
  UNDER Persons;
CREATE TABLE Students OF t_Student
  UNDER Persons;
```

At the data level those two declarations imply that instances of `Teachers` and `Students` relations are also instances of the `Persons` relation (inheritance among relations) and that instances of those relations have `name`, `ssn`, `b_date`, and `address` as attributes (inheritance among types). The query

```
SELECT name, address
FROM Teachers
WHERE salary > 2000
```

can thus be expressed.

Inheritance among types also implies method inheritance, and method overloading. Overriding and late binding are supported. Multiple inheritance is also supported.

LOBs. Object relational DBMS, finally, provide LOB types to support the storage of multimedia objects, such as documents, images, and audio messages. LOBs are semantically stored as columns of relations. Physically, however, they are stored outside the relations, typically in external files. Usually, for efficiency reasons, those external files are not manipulated under transactional control (or, at least, logging is disabled). LOBs can be either CLOBs (characters) or BLOBs (binaries). Ad hoc indexing mechanisms are exploited to efficiently handle LOBs.

The following relation declaration illustrates the specification of an attribute containing textual information and of an attribute containing an image:

```
CREATE TABLE Patients (name              CHAR(20),
                       ssn               INTEGER,
                       age               INTEGER,
                       clinical-register CLOB,
                       x-ray             BLOB);
```

CONCLUDING REMARKS

In this article, we have focused on the modeling aspects and query and data manipulation languages of OODBM and object relational DBMS. The effective support of object oriented data models and languages requires revisting and possibly extending techniques and data structures used in DBMS architectures. In the remainder of this section we briefly discuss some of those architectural issues and point out relevant references.

A first important aspect is related to the indexing techniques used to speed up query executions. The following three object-oriented concepts have an impact on the evaluation of object-oriented queries, as well as on the indexing support required.

Class Hierarchy. Unlike the relational model where a query on a relation R retrieves tuples from only R itself, an object-oriented query on a class C has two possible interpretations. In a *single-class query,* objects are retrieved from only the queried class C itself. In a *class-hierarchy query,* objects are retrieved from all the subclasses of C since any object of a subclass of C is also an object of C. The interpretation of the query type (single-class or class-hierarchy) is specified by the user. To facilitate the evaluation of such types of queries, a *class-hierarchy index* needs to support efficient retrieval of objects from a single class, as well as from all the classes in the class hierarchy.

Aggregation Hierarchy. In an object-oriented data model, a class can be defined as a nested structure of classes, giving rise to an aggregation hierarchy. An aggregation index must index object paths efficiently. Without efficient index support, the evaluation of such queries can be slow because it requires access to multiple classes.

Methods. To speed up the evaluation of object-oriented query predicates that involve methods, efficient index support is required.

A class-hierarchy index is characterized by two parameters: (1) the hierarchy of classes to be indexed and (2) the index attribute of the indexed hierarchy. There are two approaches to class-hierarchy indexing:

- Class-dimension-based approach (25,26) partitions the data space primarily on the class of an object.
- Attribute-dimension-based approach (25) partitions the data space primarily on the indexed attribute of an object.

While the class-dimension-based approach supports single-class queries efficiently, it is not effective for class-hierarchy queries due to the need traversing multiple single-class indexes. On the other hand, the attribute-dimension-based approach generally provides efficient support for class-hierarchy queries on the root class (i.e., retrieving objects of all the indexed classes), but is inefficient for single-class queries or class-hierarchy queries on a subhierarchy of the indexed class hierarchy, because it may need to access many irrelevant leaf nodes of the single index structure. To support both types of queries efficiently, the index must support both ways of data partitioning (27). However, this is not a simple or direct application of multidimensional indexes, since total ordering of classes is not possible and hence partitioning along the class dimension is problematic. A second important issue in indexing techniques is related to aggregation hierarchies and navigational accesses along these hierarchies. Navigational access is based on traversing object references; a typical example is represented by graph traversal. Navigations from one object in a class to objects in other classes in a class aggregation hierarchy are essentially expensive pointer chasing operations. To support navigations efficiently, indexing structures that enable fast path instantiation have been developed, including the multi-index technique, the nested index, the path index, and the join hierarchy index. In practice, many of these structures are based on precomputing traversals along aggregation hierarchies. The major problem of many of such indexing techniques is related to update operations that may require access to several objects in order to determine the index entries that need update. To reduce update overhead and yet maintain the efficiency of path indexing structures, paths can be broken into subpaths which are then indexed separately (28,29). The proper splitting and allocation is highly dependent on the query and update patterns and frequencies. Therefore adequate index allocation tools should be developed to support the optimal index allocation. Finally, a last issue to discuss is related to the use of user-defined methods into queries. The execution of a query involving such a method may require the execution of such a method for a large number of instances. Because a method can be a general program, the query execution costs may become prohibitive. Possible solutions, not yet fully investigated, are based on method precomputation; such approaches, however, make object updates rather expensive. We refer the reader to Ref. 30 for an extensive discussion on indexing techniques for OODBMS.

Another important issue, related to performance, is query optimization. Since most object-oriented queries only require implicit joints through aggregation hierarchies, the efficient support of such join is important. Therefore, proposed query execution strategies have focused on efficient traversal of aggregation hierarchies. Because aggregation hierarchies can be represented as graphs, and a query can be seen in a visit of a portion of such a graph, traversal strategies can be formalized as strategies for visiting nodes in a graph. The main methods proposed for such visits include: forward traversal, reverse traversal, and midex traversal. They differ with respect to the order according to which the nodes involved in a given query are visited. A second dimension in query processing strategies concerns how instances from the visited class are retrieved. The two main strategies are the nested loop and the sort domain. Each of those strategies can be combined with each node traversal strategy, resulting in a wide spectrum of strategies. We refer the reader to Ref. 1 for an extensive discussion on query execution strategies and related cost models.

Other relevant issues that we do not discuss here include access control mechanisms, versioning models, schema evolutions, benchmarks, concurrency control and transaction management mechanisms. We refer the interested reader to (1).

BIBLIOGRAPHY

1. E. Bertino and L. D. Martino, *Object-Oriented Database Systems—Concepts and Architecture,* Reading, MA: Addison-Wesley, 1993.
2. R. Cattel, *Object Data Management—Object-Oriented and Extended Relational Database Systems,* Reading, MA: Addison-Wesley, 1991.
3. W. Kim and F. H. Lochovsky, *Object-Oriented Concepts, Databases, and Applications,* Reading, MA: Addison-Wesley, 1989.
4. O. J. Dahl and K. Nygaard, Simula: An Algol based Simulation Language, *Commun. ACM,* **9**: 671–678, 1966.
5. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation,* Reading, MA: Addison-Wesley, 1983.
6. B. Meier, *Object Oriented Software Construction,* Englewood Cliffs, NJ: Prentice-Hall, 1988.
7. L. G. DeMichiel and R. P. Gabriel, The common lisp object system: An overview, *Proc. 1st Eur. Conf. Object-Oriented Programming,* 1987.
8. B. Stroustup, *The C++ Programming Language,* Reading, MA: Addison-Wesley, 1986.
9. K. Arnold and J. Goslin, *The Java Programming Language,* Reading, MA: Addison-Wesley, 1996.
10. P. Wegner, Dimensions of object-based language design. In N. Meyrowitz, (ed.), *Proc. 2nd Int. Conf. Object-Oriented Programming: Syst., Languages, Appl.,* 1987, pp. 168–182.
11. P. Chen, The entity–relationship model—towards a unified view of data. *ACM Trans. Database Syst.,* **1** (1): 9–36, 1976.
12. C. Beeri, Formal models for object oriented databases. In W. Kim et al. (ed.), *Proc. 1st Int. Conf. Deductive Object-Oriented Databases,* 1989, pp. 370–395.
13. L. Cardelli and P. Wegner, On understanding types, data abstraction and polimorphism, *Comput. Surv.,* **17**: 471–522, 1985.

14. M. Atkinson et al., The object-oriented database system manifesto. In W. Kim et al. (eds.), *Proc. 1st Int. Conf. Deductive Object-Oriented Databases,* 1989, pp. 40–57.
15. E. Bertino et al., Object-oriented query languages: The notion and the issues, *IEEE Trans. Knowl. Data Eng.,* **4**: 223–237, 1992.
16. ServioLogic Development Corporation, *Programming in OPAL,* 1990, Version 2.0.
17. F. Bancilhon, Object-oriented database systems, *Proc. 7tn ACM SIGACT-SIGMOD-SIGART Symp. Principles Database Syst.,* 1988.
18. A. Nori, Object relational DBMSs, *22nd Int. Conf. Very Large Data Bases—Tutorial,* 1996.
19. D. Chamberlin, *Using the New DB2—IBM's Object-Relational Database System,* San Mateo, CA: Morgan-Kaufmann, 1996.
20. W. Kim, UniSQL/X Unified Relational and Object-Oriented Database System, *Proc. ACM SIGMOD Int. Conf. Manage. Data,* 1994, p. 481.
21. Illustra Information Technologies, Oakland, California, *Illustra User's Guide,* Release 2.1.
22. ORACLE 7.0, *SQL Language—Reference Manual,* 1992.
23. SYBASE Inc., Berkley, California, *Transact-SQL User's Guide for Sybase,* Release 10.0.
24. J. Melton and A. R. Simon, *Understanding the New SQL: A Complete Guide,* San Mateo, CA: Morgan-Kaufmann, 1993.
25. W. Kim, K. C. Kim, and A. Dale, Indexing techniques for object-oriented databases. In W. Kim and F. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications,* Reading, MA: Addison-Wesley, 1989, pp. 371–394.
26. C. C. Low, B. C. Ooi, and H. Lu, H-trees: A dynamic associative search index for OODB, *Proc. 1992 ACM SIGMOID Int. Conf. Manage. Data,* 1992, pp. 134–143.
27. C. Y. Chan, C. H. Goh, and B. C. Ooi, Indexing OODB instances based on access proximity, *Proc. 13th Int. Conf. Data Eng.,* 1997, pp. 14–21.
28. E. Bertino, On indexing configuration in object-oriented databases, *VLDB J.* **3** (3): 355–399, 1994.
29. Z. Xie and J. Han, Join index hierarchy for supporting efficient navigation in object-oriented databases, *Proc. 20th Int. Conf. Very Large Data Bases,* 1994, pp. 522–533.
30. E. Bertino et al., *Indexing Techniques for Advanced Database Systems,* Norwell, MA: Kluwer, 1997.
31. R. Breitl et al., The GemStone Data Management System. In W. Kim and F. H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications,* Reading, MA: Addison-Wesley, 1989, pp. 283–308.
32. D. H. Fishman et al., Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications,* Reading, MA: Addison-Wesley. 1989, pp. 219–250.
33. F. Bancilhon, C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System: The Story of O_2*, San Mateo, CA: Morgan-Kaufmann, 1992.
34. O. Deux et al., The Story of O_2, *IEEE Trans. Knowledge Data Eng.,* **2**: 91–108, 1990.
35. W. Kim et al., Features of the ORION object-oriented database system. In W. Kim and F. H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications,* Addison-Wesley, 1989, pp. 251–282.
36. W. Kim, *Introduction to Object-Oriented Databases,* Cambridge, MA: The MIT Press, 1990.
37. *ObjectStore Reference Manual,* 1990, Burlington, MA: Object Design Inc.
38. R. Agrawal and N. Gehani, ODE (object database and environment): The language and the data model, *Proc. ACM SIGMOD Int. Conf. Manage. Data,* 1989, pp. 36–45.
39. R. Cattel, *The Object Database Standard: ODMG-93,* San Mateo: Morgan-Kaufmann, 1996.

ELISA BERTINO
Università degli Studi di Milano
GIOVANNA GUERRINI
Università di Genova