Unit 3: Emerging Database Management System Technologies
Part I
(Fundamentals of Database Systems, Third Edition by Ramez Elmasri and Shamkant B. Navathe)

Object Oriented Database concepts:
11.1 Overview of Object-Oriented Concepts

The term object-oriented—abbreviated by OO or O-O—has its origins in OO programming languages, or OOPLs. Today OO concepts are applied in the areas of databases, software engineering, knowledge bases, artificial intelligence, and computer systems in general. OOPLs have their roots in the SIMULA language, which was proposed in the late 1960s. In SIMULA, the concept of a class groups together the internal data structure of an object in a class declaration. Subsequently, researchers proposed the concept of abstract data type, which hides the internal data structures and specifies all possible external operations that can be applied to an object, leading to the concept of encapsulation. The programming language SMALLTALK, developed at Xerox PARC in the 1970s, was one of the first languages to explicitly incorporate additional OO concepts, such as message passing and inheritance. It is known as a pure OO programming language, meaning that it was explicitly designed to be object-oriented. This contrasts with hybrid OO programming languages, which incorporate OO concepts into an already existing language. An example of the latter is C++, which incorporates OO concepts into the popular C programming language.
An object typically has two components: state (value) and behavior (operations). Hence, it is somewhat similar to a program variable in a programming language, except that it will typically have a complex data structure as well as specific operations defined by the programmer. Objects in an OOPL exist only during program execution and are hence called transient objects. An OO database can extend the existence of objects so that they are stored permanently, and hence the objects persist beyond program termination and can be retrieved later and shared by other programs. In other words, OO databases store persistent objects permanently on secondary storage, and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as indexing mechanisms, concurrency control, and recovery. An OO database system interfaces with one or more OO programming languages to provide persistent and shared object capabilities.

One goal of OO databases is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, OO databases provide a unique system-generated object identifier (OID) for each object.
We can compare this with the relational model where each relation must have a primary key attribute whose value identifies each tuple uniquely. In the relational model, if the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same object (for example, the object identifier may be represented as EMP_ID in one relation and as SSN in another). Another feature of OO databases is that objects may have an object structure of arbitrary complexity in order to contain all of the necessary information that describes the object. In contrast, in traditional database systems, information about a complex object is often scattered over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.

The internal structure of an object in OOPLs includes the specification of instance variables, which hold the values that define the internal state of the object. Hence, an instance variable is similar to the concept of an attribute, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some

OO models insist that all operations a user can apply to an object must be predefined. This forces a complete encapsulation of objects. This rigid approach has been relaxed in most OO data models for several reasons. First, the database user often needs to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects. Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify on the fly.

To encourage encapsulation, an operation is defined in two parts. The first part, called the signature or interface of the operation, specifies the operation name and arguments (or parameters). The second part, called the method or body, specifies the implementation of the operation. Operations can be invoked by passing a message to an object, which includes the operation name and the parameters. The object then executes the method for that operation. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations. Hence, encapsulation provides a form of data and operation independence. Another key concept in OO systems is that of type and class hierarchies and inheritance. This permits specification of new types or classes that inherit much of their structure and operations from previously defined types or classes. Hence, specification of object types can proceed systematically. This makes it easier to develop the data types of a system incrementally, and to reuse existing type definitions when creating new types of objects.

One problem in early OO database systems involved representing relationships among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships, because it is useful to identify these relationships and make them visible to users.

The ODMG 2.0 standard has recognized this need and it explicitly represents binary relationships via a pair of inverse references—that is, by placing the OIDs of related objects within the objects themselves, and maintaining referential integrity.

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

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

11.2 Object Identity, Object Structure, and Type Constructors

11.2.1 Object Identity

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

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

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

11.2.2 Object Structure

In OO databases, the state (current value) of a complex object may be constructed from other objects (or other values) by using certain type constructors. One formal way of representing such objects is to view each object as a triple (i, c, v), where i is a unique object identifier (the OID), c is a type constructor (that is, an indication of how the object state is constructed), and v is the object state (or current value). The data model will typically include several type constructors. The three most basic constructors are atom, tuple, and set. Other commonly used constructors include list, bag, and array. The atom constructor is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly. The object state v of an object (i, c, v) is interpreted based on the constructor c. If c = atom, the state (value) v is an atomic value from the domain of basic values supported by the system. If c = set, the state v is a set of object identifiers , which are the OIDs for a set of objects that are typically of the same type. If c = tuple, the state v is a tuple of the form , where each is an attribute name and each is an OID. If c = list, the value v is an ordered list of OIDs of objects of the same type. A list is similar to a set except that the OIDs in a list are ordered, and hence we can refer to the first, second, or object in a list. For c = array, the state of the object is a single-dimensional array of object identifiers.

The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size. The difference between set and bag is that all elements in a set must be distinct whereas a bag can have duplicate elements.
This model of objects allows arbitrary nesting of the set, list, tuple, and other constructors. The state of an object that is not of type atom will refer to other objects by their object identifiers. Hence, the only case where an actual value appears is in the state of an object of type atom.
The type constructors set, list, array, and bag are called collection types (or bulk types), to distinguish them from basic types and tuple types. The main characteristic of a collection type is that the state of the object will be a collection of objects that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The tuple type constructor is often called a structured type, since it corresponds to the struct construct in the C and C++ programming languages.

11.2.3 Type Constructors
An object definition language (ODL) that incorporates the preceding type constructors can be used to define the object types for a particular database application. The type constructors can be used to define the data structures for an OO database schema. In Section 11.3 we will see how to incorporate the definition of operations (or methods) into the OO schema. Figure 11.02 shows how we may declare Employee and Department types corresponding to the object instances shown in Figure 11.01. In Figure 11.02, the Date type is defined as a tuple rather than an atomic value as in Figure 11.01. We use the keywords tuple, set, and list for the type constructors, and the available standard data types (integer, string, float, and so on) for atomic types.

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

11.3 Encapsulation of Operations, Methods, and Persistence

The concept of encapsulation is one of the main characteristics of OO languages and systems. It is also related to the concepts of abstract data types and information hiding in programming languages. In traditional database models and systems, this concept was not applied, since it is customary to make the structure of database objects visible to users and external programs. In these traditional models, a number of standard database operations are applicable to objects of all types. For example, in the relational model, the operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to any relation in the database. The relation and its attributes are visible to users and to external programs that access the relation by using these operations.

11.3.1 Specifying Object Behavior via Class Operations
The concepts of information hiding and encapsulation can be applied to database objects. The main idea is to define the behavior of a type of object based on the operations that can be externally applied to objects of that type. The internal structure of the object is hidden, and the object is accessible only through a number of predefined operations. Some operations may be used to create (insert) or destroy (delete) objects; other

operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations. Still other operations may perform a combination of retrieval, calculation, and update. In general, the implementation of an operation can be specified in a general-purpose programming language that provides flexibility and power in defining the operations. The external users of the object are only made aware of the interface of the object type, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of the internal data structures of the object and the implementation of the operations that access these structures. In OO terminology, the interface part of each operation is called the signature, and the operation implementation is called a method. Typically, a method is invoked by sending a message to the object to execute the corresponding method. Notice that, as part of executing a method, a subsequent message to another object may be sent, and this mechanism may be used to return values from the objects to the external environment or to other objects.
For database applications, the requirement that all objects be completely encapsulated is too stringent.
One way of relaxing this requirement is to divide the structure of an object into visible and hidden attributes (instance variables). Visible attributes may be directly accessed for reading by external operators, or by a high-level query language. The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations. Most OODBMSs employ high-level query languages for accessing visible attributes.

In most cases, operations that update the state of an object are encapsulated. This is a way of defining the update semantics of the objects, given that in many OO data models, few integrity constraints are predefined in the schema. Each type of object has its integrity constraints programmed into the methods that create, delete, and update the objects by explicitly writing code to check for constraint violations and to handle exceptions. In such cases, all update operations are implemented by encapsulated operations. More recently, the ODL for the ODMG 2.0 standard allows the specification of some constraints such as keys and inverse relationships (referential integrity) so that the system can automatically enforce these constraints.

The term class is often used to refer to an object type definition, along with the definitions of the operations for that type (Note 18). Figure 11.03 shows how the type definitions of Figure 11.02 may be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition. A method (implementation) for each operation must be defined elsewhere, using a programming language. Typical operations include the object constructor operation, which is used to create a new object, and the destructor operation, which is used to destroy an object. A number of object modifier operations can also be declared to modify various attributes of an object. Additional operations can retrieve information about the object.

An operation is typically applied to an object by using the dot notation. For example, if d is a reference to a department object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted). The only exception is the constructor operation, which returns a reference to a new Department object. Hence, it is customary to have a default name for the constructor operation that is the name of the class itself, although this was not used in Figure 11.03 (Note 19). The dot notation is also used to refer to attributes of an object—for example, by writing d.dnumber or d.mgr.startdate.

11.3.2 Specifying Object Persistence via Naming and Reachability

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

the database and persist after program termination. The typical mechanisms for making an object persistent are naming and reachability.
The naming mechanism involves giving an object a unique persistent name through which it can be retrieved by this and other programs. This persistent object name can be given via a specific statement or operation in the program, as illustrated in Figure 11.04. All such names given to objects must be unique within a particular database. Hence, the named persistent objects are used as entry points to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called reachability. The reachability mechanism works by making the object reachable from some persistent object. An object B is said to be reachable from an object A if a sequence of references in the object graph lead from object A to object B. For example, all the objects in Figure 11.01 are reachable from object ; hence, if is made persistent, all the other objects in Figure 11.01 also become persistent.

If we first create a named persistent object N, whose state is a set or list of objects of some class C, we can make objects of C persistent by adding them to the set or list, and thus making them reachable from N. Hence, N defines a persistent collection of objects of class C. For example, we can define a class DepartmentSet (see Figure 11.04) whose objects are of type set(Department).

Suppose that an object of type DepartmentSet is created, and suppose that it is named AllDepartments and thus made persistent, as illustrated in Figure 11.04. Any Department object that is added to the set of AllDepartments by using the add_dept operation becomes persistent by virtue of its being reachable from AllDepartments. The AllDepartments object is often called the extent of the class Department, as it will hold all persistent objects of type Department. As we shall see in Chapter 12, the ODMG ODL standard gives the schema designer the option of naming an extent as part of class definition. Notice the difference between traditional database models and OO databases in this respect. In traditional database models, such as the relational model or the EER model, all objects are assumed to be persistent. Hence, when an entity type or class such as EMPLOYEE is defined in the EER model, it represents both the type declaration for EMPLOYEE and a persistent set of all EMPLOYEE objects. In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) or list(EMPLOYEE) whose value is the collection of references to all persistent EMPLOYEE objects, if this is desired, as illustrated in Figure 11.04 (Note 21). In fact, it is possible to define several persistent collections for the same class definition, if desired. This allows transient and persistent objects to follow the same type and class declarations of the ODL and the OOPL.

11.4 Type Hierarchies and Inheritance

Another main characteristic of OO database systems is that they allow type hierarchies and inheritance.
Type hierarchies in databases usually imply a constraint on the extents corresponding to the types in the hierarchy.

11.4.1 Type Hierarchies and Inheritance

In most database applications, there are numerous objects of the same type or class. Hence, OO databases must provide a capability for classifying objects based on their type, as do other database systems. But in OO databases, a further requirement is that the system permit the definition of new types based on other predefined types, leading to a type (or class) hierarchy.
Typically, a type is defined by assigning it a type name and then defining a number of attributes (instance variables) and operations (methods) for the type. In some cases, the attributes and operations are together called functions, since attributes resemble functions with zero arguments. A function name can be used to refer to the value of an attribute or to refer to the resulting value of an operation (method). In this section, we use the term function to

refer to both attributes and operations of an object type, since they are treated similarly in a basic introduction to inheritance.
A type in its simplest form can be defined by giving it a type name and then listing the names of its visible (public) functions. When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:
TYPE_NAME: function, function, . . . , function

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

PERSON: Name, Address, Birthdate, Age, SSN
In the PERSON type, the Name, Address, SSN, and Birthdate functions can be implemented as stored attributes, whereas the Age function can be implemented as a method that calculates the Age from the value of the Birthdate attribute and the current date.
The concept of subtype is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which we shall call the supertype. For example, suppose that we want to define two new types EMPLOYEE and STUDENT as follows:

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

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

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

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

In general, a subtype includes all of the functions that are defined for its supertype plus some additional functions that are specific only to the subtype. Hence, it is possible to generate a type hierarchy to show the supertype/subtype relationships among all the types declared in the system. As another example, consider a type that describes objects in plane geometry, which may be defined as follows:
GEOMETRY_OBJECT: Shape, Area, ReferencePoint

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

RECTANGLE subtype-of GEOMETRY_OBJECT: Width, Height

TRIANGLE subtype-of GEOMETRY_OBJECT: Side1, Side2, Angle
CIRCLE subtype-of GEOMETRY_OBJECT: Radius
Notice that the Area operation may be implemented by a different method for each subtype, since the procedure for area calculation is different for rectangles, triangles, and circles. Similarly, the attribute ReferencePoint may have a different meaning for each subtype; it might be the center point for RECTANGLE and CIRCLE objects, and the vertex point between the two given sides for a TRIANGLE object. Some OO database systems allow the renaming of inherited functions in different subtypes to reflect the meaning more closely. An alternative way of declaring these three subtypes is to specify the value of the Shape attribute as a condition that must be satisfied for objects of each subtype:

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

Here, only GEOMETRY_OBJECT objects whose Shape='rectangle' are of the subtype RECTANGLE, and similarly for the other two subtypes. In this case, all functions of the GEOMETRY_OBJECT supertype are inherited by each of the three subtypes, but the value of the Shape attribute is restricted to a specific value for each.
Notice that type definitions describe objects but do not generate objects on their own. They are just declarations of certain types; and as part of that declaration, the implementation of the functions of each type is specified. In a database application, there are many objects of each type. When an object is created, it typically belongs to one or more of these types that have been declared. For example, a circle object is of type CIRCLE and GEOMETRY_OBJECT (by inheritance). Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are meaningful to the database application.

11.5 Complex Objects

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

11.5.1 Unstructured Complex Objects and Type Extensibility

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

The DBMS software does not have the capability to directly process selection conditions and other operations based on values of these objects, unless the application provides the code to do the comparison operations needed for the selection. In an OODBMS, this can be accomplished by defining a new abstract data type for the uninterpreted objects and by providing the methods for selecting, comparing, and displaying such objects. For

example, consider objects that are two-dimensional bitmap images. Suppose that the application needs to select from a collection of such objects only those that include a certain pattern. In this case, the user must provide the pattern recognition program as a method on objects of the bitmap type. The OODBMS then retrieves an object from the database and runs the method for pattern recognition on it to determine whether the object includes the required pattern.

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

11.5.2 Structured Complex Objects

A structured complex object differs from an unstructured complex object in that the object's structure is defined by repeated application of the type constructors provided by the OODBMS. Hence, the object structure is defined and known to the OODBMS. As an example, consider the DEPARTMENT object shown in Figure 11.01. At the first level, the object has a tuple structure with six attributes:
DNAME, DNUMBER, MGR, LOCATIONS, EMPLOYEES, and PROJECTS. However, only two of these attributes—namely, DNAME and DNUMBER—have basic values; the other four have complex values and hence build the second level of the complex object structure. One of these four (MGR) has a tuple structure, and the other three (LOCATIONS, EMPLOYEES, PROJECTS) have set structures. At the third level, for a MGR tuple value, we have one basic attribute (MANAGERSTARTDATE) and one attribute (MANAGER) that refers to an employee object, which has a tuple structure. For a LOCATIONS set, we have a set of basic values, but for both the EMPLOYEES and the PROJECTS sets, we have sets of tuple-structured objects.

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

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

11.6 Other Objected-Oriented Concepts

11.6.1 Polymorphism (Operator Overloading)

Another characteristic of OO systems is that they provide for polymorphism of operations, which is also sometimes referred to as operator overloading. This concept allows the same operator name or symbol to be bound to two or more different implementations of the operator, depending on the type of objects to which the operator is applied. A simple example from programming languages can illustrate this concept. In some languages, the operator symbol "+" can mean different things when applied to operands (objects) of different types. If the operands of "+" are of type integer, the operation invoked is integer addition. If the operands of "+" are of type floating point, the operation invoked is floating point addition. If the operands of "+" are of type set, the operation invoked is set union. The compiler can determine which operation to execute based on the types of operands supplied.
In OO databases, a similar situation may occur. We can use the GEOMETRY_OBJECT example discussed in Section 11.4 to illustrate polymorphism in OO databases. Suppose that we declare
GEOMETRY_OBJECT and its subtypes as follows:

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

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

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

11.6.2 Multiple Inheritance and Selective Inheritance

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

subtype ENGINEERING_MANAGER. It is possible, however, that both ENGINEER and MANAGER inherit Salary from the same supertype (such as EMPLOYEE) higher up in the lattice. The general rule is that if a function is inherited from some common supertype, then it is inherited only once. In such a case, there is no ambiguity; the problem only arises if the functions are distinct in the two supertypes.

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

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

11.6.3 Versions and Configurations

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

Notice that there may be more than two versions of an object. For example, consider two programmers working to update the same software module concurrently. In this case, two versions, in addition to the original module, are needed. The programmers can update their own versions of the same software module concurrently. This is often referred to as concurrent engineering. However, it eventually becomes necessary to merge these two versions together so that the new (hybrid) version can include the changes made by both programmers. During merging, it is also necessary to make sure that their changes are compatible. This necessitates creating yet another version of the object: one that is the result of merging the two independently updated versions.

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

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