Chapter 14

DISTRIBUTED OBJECT DATABASE MANAGEMENT SYSTEMS

In this chapter, we relax another one of the fundamental assumptions we made in Chapter 1 — namely that the system implements the relational data model. Database technology is rapidly evolving toward the support of new applications. Relational databases have proven to be very successful in supporting business data processing applications. However, there is now an important class of applications, commonly referred to as "advanced database applications," that exhibit pressing needs for database management. Examples include computer-aided design (CAD), office information systems (OIS), multimedia information systems, and artificial intelligence (AI). For these applications, object database management systems (object DBMSs) are considered to be more suitable due to the following characteristics [Özsu et al., 1994b]:

1. These advanced applications require explicit storage and manipulation of more abstract data types (e.g., images, design documents) and the ability for the users to define their own application-specific types. Therefore, a rich type system supporting user-defined abstract types is required. Relational systems deal with a single object type, a relation, whose attributes come from simple and fixed data type domains (e.g., numeric, character, string, date). There is no support for explicit definition and manipulation of application-specific types.

2. The relational model structures data in a relatively simple and flat manner. Representing structural application objects in the flat relational model results in the loss of natural structure that may be important to the application. For example, in engineering design applications, it may be preferable to explicitly represent that a vehicle object contains an engine object. Similarly, in a multimedia information system, it is important to note that a hyperdocument object contains a particular video object and a captioned text object. This "containment" relationship between applica-

tion objects is not easy to represent in the relational model, but is fairly straightforward in object models by means of *composite objects* and *complex objects*, which we discuss shortly.

3. Relational systems provide a declarative and (arguably) simple language for accessing the data – SQL. Since this is not a computationally complete language, complex database applications have to be written in general programming languages with embedded query statements. This causes the well-known "impedance mismatch" [Copeland and Maier, 1984] problem, which arises because of the differences in the level of abstraction between the relational languages and the programming languages with which they interact. The concepts and types of the query language, typically set-at-a-time, do not match with those of the programming language, which are typically record-at-a-time. In an object system, complex database applications may be written entirely in a single object database programming language [Atkinson and Buneman, 1987]. Of course, this may change as SQL develops toward a computationally complete language.

14.1 FUNDAMENTAL OBJECT CONCEPTS AND OBJECT MODELS

An object DBMS is a system that uses an "object" as the fundamental modeling and access primitive. Beyond this fairly straightforward statement, what constitutes an object DBMS is a topic of considerable discussion [Atkinson et al., 1989], [Stonebraker et al., 1990a]. Contrary to the relational model, there is no universally accepted and formally specified object model. There are a number of features that are common to most model specifications, but the exact semantics of these features are different in each model. Even the feasibility of defining an object model, in the same sense as the relational model, has been questioned [Maier, 1989]. Some standard object model specifications are emerging as part of language standards (e.g., Object Data Management Group's (ODMG) model [Cattell, 1997] or SQL-3 [Melton, 1998]), but these are not widely adopted. In the remainder of this section, we will review some of the design issues and alternatives in defining an object model.

14.1.1 Object

As indicated above, all object DBMSs are built around the fundamental concept of an *object*. An object represents a real entity in the system that is being modeled. Most simply, it is represented as a pair (OID, state), in which OID is the object identity and the corresponding state is some representation of the current state of the object. Object identity [Khoshafian and Copeland, 1986] is an invariant property of an object which permanently distinguishes it logically and physically from all other objects, regardless of its state. This enables referential object sharing [Khoshafian and Valduriez, 1987], which is the basis for supporting

composite and complex (i.e., graph) structures (see Section 14.1.3). In some models, OID equality is the only comparison primitive; for other types of comparisons, the type definer is expected to specify the semantics of comparison. In other models, two objects are said to be *identical* if they have the same OID, and *equal* if they have the same state.

Many object models start to diverge at the definition of the *state*. Some object models define state as either an atomic value or a constructed value (e.g., tuple or set). Let D be the union of the system-defined domains (e.g., domain of integers) and of user-defined abstract data type (ADT) domains (e.g., domain of companies), let I be the domain of identifiers used to name objects, and let A be the domain of attribute names. A *value* is defined as follows:

1. An element of D is a value, called an *atomic value*.
2. $[a_1 : v_1, \ldots, a_n : v_n]$, in which a_i is an element of A and v_i is either a value or an element of I, is called a *tuple value*. [ ] is known as the tuple constructor.
3. $\{v_1, \ldots, v_n\}$, in which v_i is either a value or an element of I, is called a *set value*. { } is known as the set constructor.

These models consider object identifiers as values (similar to pointers in programming languages). Set and tuple are data constructors that we consider essential for database applications. Other constructors, such as list or array, could also be added to increase the modeling power.

Example 14.1

Consider the following objects:

$(i_1, 231)$
$(i_2, S70)$
$(i_3, \{i_6, i_{11})$
$(i_4, \{1, 3, 5\})$
$(i_5, [LF: i_7, RF: i_8, LR: i_9, RR: i_10])$

Objects i_1 and i_2 are atomic objects and i_3 and i_4 are constructed objects, i_3 is the OID of an object whose state consists of a set. The same is true of i_4. The difference between the two is that the state of i_4 consists of a set of values, while that of i_3 consists of a set of OIDs. Thus, object i_3 references other objects. By considering object identifiers (e.g., i_6) as values in the object model, arbitrarily complex objects may be constructed. Object i_5 has a tuple valued state consisting of four attributes (or instance variables), the values of each being another object.

Contrary to values, objects support a well-defined update operation that changes the object state without changing the object identity. This is analogous to updates

in imperative programming languages in which object identity is implemented by main memory pointers. However, object identity is more general than pointers in the sense that it persists following the program termination. Another implication of object identity is that objects may be shared without incurring the problem of data redundancy. We will discuss this further in Section 14.1.3.

Example 14.2

Consider the following objects:

(i_1, Volvo)
(i_2, [name: John, mycar: i_1])
(i_3, [name: Mary, mycar: i_1])

John and Mary share the object denoted by mycar (they both own Volvo cars). Changing the value of object i_1 from "Volvo" to "Chevrolet" is automatically seen by both objects i_2 and i_3.

The above discussion captures the structural aspects of a model – the state is represented as a set of *instance variables* (or *attributes*) which are values. The behavioral aspects of the model are captured in *methods*, which define the allowable operations on these objects and are used to manipulate them. Methods represent the behavioral side of the model because they define the legal behaviors that the object can assume. A classical example is that of an elevator [Jones, 1979]. If the only two methods defined on an elevator object are "up" and "down", they together define the behavior of the elevator object: it can go up or down, but not sideways, for example.

Other object models take a more behavioral approach [Dayal, 1989], [Özsu et al., 1995a]. In these models, there are no instance variables or methods; there are only behaviors, which are applied to objects to achieve a certain result. The behaviors may be implemented either by stored functions or by computed functions, but this distinction is not visible at the user level. Consequently, the "state" of an object is defined by the results returned by the application of these behaviors.

Example 14.3

Consider Example 14.2. In the relational model, to achieve the same purpose, one would typically set the value of attribute `mycar` to "Volvo", which would require both tuples to be updated when it changes to "Chevrolet". To reduce redundancy, one can still represent i_1 as a tuple in another relation and reference it from i_1 and i_2 using foreign keys. Recall that this is the basis of 3NF and BCNF normalization. In this case, the elimination of redundancy requires, in the relational model, normalization of relations. However, i_1 may be a structured object whose representation in a normalized relation may be awkward. In this case, we cannot assign it as the value of the mycar attribute even if we accept the redundancy, since the relational model requires attribute values to be atomic.

14.1.2 Abstract Data Types

Abstract data types (ADTs) have long been used in programming languages [Guttag, 1977], and more recently in relational databases [Stonebraker et al., 1983b], [Osborn and Heaven, 1986], and [Gardarin and Valduriez, 1989]. In fact, the introduction of abstract data types into relational systems is the basis of the newly emerging object-relational DBMSs [Stonebraker and Brown, 1999]. An *abstract data type* (usually simply called a *type*[1]) is a template for all objects of that type. In this case, we don't make a distinction between primitive system objects (i.e., values), structural (tuple or set) objects, and user-defined objects. An ADT describes the type of data by providing a domain of data with the same structure, as well as operations (also called methods) applicable to elements of that domain. The abstraction capability of ADTs, commonly referred to as *encapsulation*, hides the implementation details of the operations, which can be written in a general-purpose programming language. Thus, each ADT is identifiable to the "outside world" by the properties that it supports. In traditional object models, the properties consist of instance variables that reflect the state of the objects, and methods that define the operations that can be performed on objects of this type. In other models, these properties are abstracted as behaviors. This is similar to the primitive operations (e.g., addition) associated with primitive types (e.g., integer) provided by the programming language. In general, several standard ADT operations, such as conversion for input-output, are mandatory. The user of the ADT only sees the interface data structure and the interface operation names with their associated input and output types.

Example 14.4

In this chapter we will use an example that demonstrates the power of object models. We will model a car that consists of various parts (engine, bumpers, tires) and will store other information such as make, model, serial number, etc. The type definition of `Car` can be as follows (not using any particular syntax):

```
type Car
    attributes
        engine : Engine
        bumpers : {Bumper}
        tires : [LF: Tire, RF: Tire, LR: Tire, RR: Tire]
        make : Manufacturer
        model : String
        year : Date
        serial_no : String
        capacity : Integer methods
        age: Real
```

[1] We will use these terms interchangeably in the remainder of the text.

The type definition specifies that Car has eight attributes and one method. Four of the attributes (model, year, serial_no, capacity) are value-based, while the others (engine, bumpers, tires and make) are object-based (i.e., have other objects as their values). Attribute bumpers is set-valued (i.e., uses the set constructor), and attribute tires is tuple-valued where the left front (LF), right front (RF), left rear (LR) and right rear (RR) tires are individually identified. Incidentally, we follow a notation where the attributes are lower case and types are capitalized. Thus, `engine` is an attribute and `Engine` is a type in the system.

One more point needs to be made about this example. The method `age` takes the system date, and the `year` attribute value and calculates the date. However, since both of these arguments, are internal to the object, they are not shown in the type definition, which is the interface for the user. If there was a method that required the users to provide an external argument, then it would have appeared in the method signature.

The interface data structure of an ADT may be arbitrarily complex or large. For example, `Car` ADT has an operation `age` which takes today's date and the manufacturing date of a car and calculates its age; it may also have more complex operations that, for example, calculate a promotional price based on the time of year. Similarly, a long document with a complex internal structure may be defined as an ADT with operations specific to document manipulation. ADTs provide two major advantages. First, the primitive types provided by the system can easily be extended with user-defined types. Since there are no inherent constraints on the notion of relational domain, such extensibility can be incorporated in the context of the relational model [Osborn and Heaven, 1986]. Second, ADT operations capture parts of the application programs which are more closely associated with data. Therefore, an object model with ADTs allows modeling of both data and operations at the same time. This does not imply, however, that operations are stored with the data. They may be stored in an operation library, similar to the data directory.

14.1.3 Composition (Aggregation)

In the examples we have discussed so far, some of the instance variables have been value-based (i.e., their domains are simple values), such as the `model` and `year` in Example 14.3, while others are object-based, such as the `make` attribute, whose domain is the set of objects that are of type `Manufacturer` In this case, the `Car` type is a *composite type* and its instances are referred to as *composite objects*. Composition is one of the most powerful features of object models. It allows sharing of objects, commonly referred to as *referential sharing, since* objects "refer" to each other by their OIDs as values of object-based attributes.

Example 14.5

Let us revise Example 14.3 as follows. Assume that c_1 is one instance of Car type which is defined in Example 14.3. If the following is true:

(i_2, [name: John, mycar: c_1])
(i_3, [name: Mary, mycar: c_1])
then this indicates that John and Mary own the same car.

A restriction on composite objects results in *complex objects*. The difference between a composite and a complex object is that the former allows referential sharing while the latter does not[2]. For example, `Car` type may have an attribute whose domain is type `Tire`. It is not natural for two instances of type `Car`, c_1 and c_2, to refer to the same set of instances of `Tire`, since one would, not expect in real life for tires to be used on multiple vehicles at the same time. This distinction between composite and complex objects is not always made, but it is an important one.

The composite object relationship between types can be represented by a *composition* (*aggregation*) *graph* (or *composition* (*aggregation*) *hierarchy* in the case of complex objects). There is an edge from instance variable I of type T_1 to type T_2 if the domain of I is T_2. The composition graphs give rise to a number of issues that we will discuss in the upcoming sections.

14.1.4 Class

Most of the current object DBMSs do not distinguish between a type and a class. This is a consequence of adopting the type systems of object programming languages such as Smalltalk and C++, which support only the concept of a class. In these systems, a *class* represents both a template for all common objects (i.e., serves as a type) and the grouping of these common objects (i.e., the extent). In this case, the database schema consists of a set of class definitions with the relationships among them (which we will discuss in Section 14.1.6).

Conceptually, however, there is a difference between a type and a class. A type is a template for all objects of that type, whereas a class is a grouping of all object instances of a given type. Some systems make this distinction explicit. In a sense, a type corresponds to a relation schema in relational databases, whereas a class corresponds to a populated relation instance. In other words, a class has an *extent* which is the collection of all objects of the type associated with the class.

14.1.5 Collection

A *collection* is a user-defined grouping of objects. Most conventional systems provide either the class construct or the collection construct. However, it can be argued that both classes and collections are useful, and that they should be supported in an object model [Beeri, 1990]. Collections provide for a clear closure semantics of the query, models and facilitate definition of user views.

[2]This distinction between composite and complex objects is not always made, and the term "composite object" is used to refer to both. Some authors reverse the definition between composite and complex objects. We will use the terms as defined here consistently in this chapter.

In systems where both classes and collections are supported, a *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. First, object creation may not occur through a collection; object creation occurs only through classes. This means that collections only form user-defined groupings of existing objects. Second, an object may exist in any number of collections, but is a member of only one class[3]. Third, the management of classes is *implicit*, in that the system automatically maintains classes based on the type lattice, whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, a class groups the entire extension of a single type (*shallow extent*), along with the extensions of its subtypes (*deep extent*). Therefore, the elements of a class are homogeneous with respect to subtyping and inheritance (see the following section). A collection is heterogeneous in that it can contain objects of types unrelated by subtyping.

14.1.6 Subtyping and Inheritance

Object systems provide extensibility by allowing user-defined types to be defined and managed by the system. This is accomplished in two ways: by the definition of types using type constructors or by the definition of types based on existing primitive types through the process of *subtyping*. Subtyping is based on the *specialization* relationship among types. A type `A` is a *specialization* of another type B if its interface is a superset of B's interface. Thus, a specialized type is more defined (or more specified) than the type from which it is specialized. A type may be a specialization of a number of types; it is explicitly specified as a *subtype* of a subset of them. Some object models require that a type is specified as a subtype of only one type, in which case the model supports *single subtyping*; others allow *multiple subtyping*, where a type may be specified as a subtype of more than one type. Subtyping and specialization indicate an `is-a` relationship between types. In the above example, `A is-a B`, resulting in *substitutability*: an instance of a subtype (`A`) can be substituted in place of an instance of any of its *supertypes* (`B`) in any expression.

Besides enabling extensibility, subtyping also gives rise to a type system that forms the database schema. In many cases, there is a single root of the type system, which is the least specified type. If only single subtyping is allowed, as in Smalltalk [Goldberg and Robson, 1983], the type system is a tree. If multiple subtyping is supported, the type system forms a graph (a semilattice, to be precise). Some systems also define a most specified type, which forms the bottom of a full lattice. It is not necessary, however, for the type system to be rooted at a single type. C++ [Stroustrup, 1986], for example, permits multiple roots, resulting in a type system with multiple graphs. In these graphs/trees, there is an edge from type (class) A to type (class) B if `A` is a subtype of `B`.

A type lattice establishes the database schema in object databases. It enables one to model the common properties and differences among types in a concise manner.

[3]We ignore subtyping and inheritance for the time being, which raise the possibility for an object to be a member of multiple classes through the deep extent of a class. This topic will be discussed later.

Example 14.6

Consider the `Car` type we defined earlier. A car can be modeled as a special type of Vehicle. Thus, it is possible to define `Car` as a subtype of `Vehicle` whose other subtypes may be `Motorcycle`, `Truck`, and `Bus`. In this case, `Vehicle` would define the common properties of all of these:

```
type Vehicle as Object
    attributes
        engine : Engine
        make : Manufacturer
        model : String
        year : Date
        serial_no : String
    methods
        age : Real
```

`Vehicle` is defined as a subtype of `Object` which we assume is the root of the type lattice. It is defined with five attributes and one method which takes the date of manufacture and today's date (both of which are of system-defined type `Date`) and returns a real value. Obviously, `Vehicle` is a generalization of `Car` that we defined in Example 14.3. `Car` can now be defined as follows:

```
type Car as Vehicle
    attributes
        bumpers : {Bumper}
        tires : [LF: Tire, RF: Tire, LR: Tire, RR: Tire]
        capacity : Integer
```

Even though `Car` is defined with only two attributes, its interface is the same as the definition given in Example 14.3. This is because `Car is-a Vehicle`, and therefore "inherits" the attributes and methods of `Vehicle`.

Declaring a type to be a subtype of another results in *inheritance*. Inheritance allows reuse. A subtype may inherit either the behavior of its subtype, or its implementation, or both. We talk of single inheritance and multiple inheritance based on the subtype relationship between the types.

14.2 OBJECT DISTRIBUTION DESIGN

Recall from Chapter 5 that the two important aspects of distribution design are fragmentation and allocation. Distribution design in the object world brings new complexities. Conceptually, objects encapsulate methods together with state. In reality, methods are implemented on types and shared by all instance objects of

that type. Therefore, the location of objects with respect to their types becomes an issue. For the same reason, partitioning classes is also difficult.

In this section we consider the analog, in object databases, of the distribution design problem introduced in Chapter 5 by considering fragmentation and allocation within the context of object models. In this discussion we assume an object model that does not distinguish between a class and a type. Distribution design in the object world brings new complexities due to the encapsulation of methods together with object state. This causes problems because methods are implemented on types and shared by all instance objects of that type. Therefore, one has to decide whether fragmentation is performed only on attributes duplicating the methods with each fragment), or whether one can fragment methods as well. The location of objects with respect to their types becomes an issue, as does the type of attributes. As discussed in Section 14.1.3, the domain of some attributes may be other classes. Thus, the fragmentation of classes with respect to such an attribute may have effects on other classes. Finally, if fragmentation is performed with respect to methods as well, it is necessary to distinguish between simple methods and complex methods. Simple methods are those that do not invoke other methods, while complex ones can invoke methods of other classes.

Similar to the relational case, there are three fundamental types of fragmentation: horizontal, vertical, and hybrid [Karlapalem et al., 1994]. In addition to these two fundamental cases, derived horizontal partitioning, associated horizontal partitioning, and path partitioning indexpath partitioning have been defined [Karlapalem and Li, 1995]. Derived horizontal partitioning has similar semantics to its counterpart in relational databases, which we will discuss further in Section 14.2.1. Associated horizontal partitioning, is similar to derived horizontal partitioning except that there is no "predicate clause", like minterm predicate, constraining the object instances. Path partitioning is discussed in Section 14.2.3. In the remainder, for simplicity, we assume a class-based object model which does not distinguish between types and classes.

14.2.1 Horizontal Class Partitioning

There are analogies between horizontal fragmentation of object databases and their relational counterparts. It is possible to identify primary horizontal fragmentation in the object database case identically to the relational case. Derived fragmentation shows some differences, however. In object databases, derived horizontal fragmentation can occur in a number of ways:

1. Partitioning of a class arising from the fragmentation of its subclasses. This occurs when a more specialized class is fragmented, so the results of this fragmentation should be reflected in the more general case. Clearly, care must be taken here, because fragmentation according to one subclass may conflict with those imposed by other subclasses. Because of this dependence, one starts with the fragmentation of the most specialized class and moves up the class lattice, reflecting its effects on the superclasses.

2. The fragmentation of a complex attribute may affect the fragmentation of its containing class.

3. Fragmentation of a class based on a method invocation sequence from one class to another may need to be reflected in the design. This happens in the case of complex methods as defined above.

Let us start the discussion with the simplest case: namely, fragmentation of a class with simple attributes and methods. In this case, primary horizontal partitioning can be performed according to a predicate defined on attributes of the class. Partitioning is easy: given class C for partitioning, we create classes $C_1, \ldots, C_n$, each of which takes the instances of C that satisfy the particular partitioning predicate. If these predicates are mutually exclusive, then classes $C_1, \ldots, C_n$ are disjoint. In this case, it is possible to define $C_1, \ldots, C_n$ as subclasses of C and change C's definition to an *abstract class* – one which does not have an explicit extent (i.e., no instances of its own). Even though this significantly forces the definition of subtyping (since the subclasses are not any more specifically defined than their superclass), it is allowed in many systems.

A complication arises if the partitioning predicates are not mutually exclusive. There are no clean solutions in this case. Some object models allow each object to belong to multiple classes. If this is an option, it can be used to address the problem. Otherwise "overlap classes" need to be defined to hold objects that satisfy multiple predicates.

Example 14.7

Consider the definition of the `Engine` class that is referred to in Example 14.6:

```
Class Engine as Object
   attributes
      no_cylinder : Integer
      capacity : Real
      horsepower : Integer
```

In this simple definition of Engine, all the attributes are simple. Consider the partitioning predicates

$$p_1: \quad \text{horsepower} \leq 150$$
$$P_2: \quad \text{horsepower} > 150$$

In this case, `Engine` can be partitioned into two classes, `Engine1` and `Engine2`, which inherit all of their properties from the Engine class, which is redefined as an abstract class. The objects of `Engine` class are distributed to the `Engine1` and `Engine2` classes based on the value of their horsepower attribute value.

This primary horizontal fragmentation of classes is applied to all classes in the system that are subject to fragmentation. At the end of this process, one obtains fragmentation schemes for every class. However, these schemes do not reflect the effect of derived fragmentation as a result of subclass fragmentation (as in the example above). Thus, the next step is to produce a set of derived fragments for each superclass using the set of predicates from the previous step. This essentially requires propogation of fragmentation decisions made in the subclasses to the superclasses. The output from this step is the set of primary fragments created in step two and the set of derived fragments from step three.

The final step is to combine these two sets of fragments in a consistent way. The final horizontal fragments of a class are composed of objects accessed by both applications running only on a class and those running on its subclasses. Therefore, we must determine the most appropriate primary fragment to merge with each derived fragment of every class. Several simple heuristics could be used, such as selecting the smallest or largest primary fragment, or the primary fragment that overlaps the most with the derived fragment. But, although these heuristics are simple and intuitive, they do not capture any quantitiative information about the distributed object database. Therefore, a more precise approach has been developed that is based on an affinity measure between fragments. As a result, fragments are joined with those fragments with which they have the highest affinity.

Let us now consider horizontal partitioning of a class with object-based instance variables (i.e., the domain of some of its instance variables is another class), but all the methods are simple. In this case, the composition relationship between classes comes into effect. In a sense, the composition relationship establishes the owner-member relationship that we discussed in Chapter 5: If class C_1 has an attribute A_1 whose domain is class C_2 then C_1 is the owner and C_2 is the member. Thus, the decomposition of C_2 follows the same principles as derived horizontal partitioning, discussed in Chapter 5.

So far, we have considered fragmentation with respect to attributes only, because the methods were simple. Let us now consider complex methods which require some care. For example, consider the case where all the attributes are simple, but the methods are complex. In this case, fragmentation based on simple attributes can be performed as described above. However, for methods, it is necessary to determine, at compile time, the objects that are accessed by a method invocation. This can be accomplished with static analysis. Clearly, optimal performance will result if invoked methods are contained within the same fragment as the invoking method. Optimization requires locating objects accessed together in the same fragment because this maximizes local relevant access and minimizes local irrelevant accesses.

The most complex case is where a class has complex attributes and complex methods. In this case, the subtyping relationships, aggregation relationships and relationships of method invocations have to be considered. Thus, the fragmentation method is the union of all of the above. One goes through the classes multiple times, generating a number of fragments, and then uses an affinity-based method to merge them.

14.2.2 Vertical Class Partitioning

Vertical fragmentation is considerably more complicated. Given a class C, fragmenting it vertically into $C_1, \ldots, C_m$ produces a number of classes, each of which contains some of the attributes and some of the methods. Thus, each of the fragments is less defined than the original class. Issues that must be addressed include the subtyping relationship between the original class superclasses and subclasses and the fragment classes, the relationship of the fragment classes among themselves, and the location of the methods. If all the methods are simple, then methods can be partitioned easily. However, when this is not the case, the location of these methods becomes a problem.

Adaptations of the affinity-based relational vertical fragmentation approaches have been developed for object databases [Ezeife and Barker, 1998], [Ezeife and Barker, 1995]. However, the break-up of encapsulation during vertical fragmentation has created significant doubts as to the suitability of vertical fragmentation in object DBMSs. We refer the reader to the above-cited literature for further information.

14.2.3 Path Partitioning

The composition graph presents a representation for composite objects. For many applications, it is necessary to access the complete composite object. Path partitioning is a concept describing the clustering of all the objects forming a composite object into a partition. A path partition consists of grouping the objects of all the domain classes that correspond to all the instance variables in the subtree rooted at the composite object.

A path partition can be represented as a hierarchy of nodes forming a structural index. Each node of the index points to the objects of the domain class of the component object. The index thus contains the references to all the component objects of a composite object, eliminating the need to traverse the class composition hierarchy. The instances of the structural index are a set of OIDs pointing to all the component objects of a composite class. The structural index is an orthogonal structure to the object database schema, in that it groups all the OIDs of component objects of a composite object as a structured index class.

14.2.4 Class Partitioning Algorithms

The main issue in class partitioning is to improve the performance of user queries and applications by reducing the irrelevant data access. Thus, class partitioning is a logical database design technique which restructures the object database schema based on the application semantics. It should be noted that class partitioning is more complicated than relation fragmentation, and is also NP-complete. The algorithms for class partitioning are based on affinity-based and cost-driven approaches.

Affinity-based Approach

As covered in Section 5.3.2, affinity among attributes is used to vertically fragment relations. Similarly, affinity among instance variables and methods, and affinity among multiple methods can be used for horizontal and vertical class partitioning. Ezeife and Barker [1994, 1995] developed horizontal and vertical class partitioning algorithms based on classifying instance variables and methods as being either simple or complex. A complex instance variable is an object-based instance variable and is part of the class composition hierarchy. Malinowski and Chakravarthy [1997] extended the partition evaluator for relational databases to object-oriented databases, and the algorithm they propose uses exhaustive enumeration. Karlapalem, et al. [1996a] developed a method induced partitioning scheme, which applies the method semantics and appropriately generates fragments that match the methods data requirements.

Cost-Driven Approach

Though the affinity-based approach provides "intuitively" appealing partitioning schemes, it has been shown [Fung et.al., 1997] that these partitioning schemes do not always result in the greatest reduction of disk accesses required to process a set of applications. Therefore, a cost model for the number of disk accesses for processing both queries [Fung et al., 1997] and methods [Fung et al., 1996] on an object-oriented database has been developed. Further, an heuristic "hill-climbing" approach which uses both the affinity approach (for initial solution) and the cost-driven approach (for further refinement) has been proposed [Fung et al., 1996]. This work also develops structural join index hierarchies for complex object retrieval, and studies its effectiveness against pointer traversal and other approaches, such as join index hierarchies, multi-index and access support relations (see next section). Each structural join index hierarchy is a materialization of path fragment, and facilitates direct access to a complex object and its component objects.

14.2.5 Allocation

The data allocation problem for object databases involves allocation of both methods and classes. The method allocation problem is tightly coupled to the class allocation problem because of encapsulation. Therefore, allocation of classes will imply allocation of methods to their corresponding home classes. But since applications on object-oriented databases invoke methods, the allocation of methods affects the performance of applications. However, allocation of methods which need to access multiple classes at different sites is a problem which has been not yet been tackled. Four alternatives can be identified [Fang et al., 1994]:

1. **Local behavior – local object.** This is the most straightforward case and is included to form the baseline case. The behavior, the object to which it is to be applied, and the arguments are all co-located. Therefore, no special mechanism is needed to handle this case.

2. **Local behavior–remote object.** This is one of the cases in which the behavior and the object to which it is applied are located at different sites. There are two ways of dealing with this case. One alternative is to move the remote object to the site where the behavior is located. The second is to ship the behavior implementation to the site where the object is located. This is possible if the receiver site can run the code.

3. **Remote behavior–local object.** This case is the reverse of case (2).

4. **Remote function–remote argument.** This case is the reverse of case (1).

Affinity-based algorithms for static allocation of class fragments that use a graph partitioning technique have also been proposed [Bhar and Barker, 1995]. However, these algorithms do not address method allocation and do not consider the inter-dependency between methods and classes. The issue has been addressed by means of an iterative solution for methods and class allocation [Bellatreche et al., 1998a].

14.2.6 Replication

Replication adds a new dimension to the design problem. Individual objects, classes of objects, or collections of objects (or all) can be units of replication. Undoubtedly, the decision is at least partially object-model dependent. Whether or not type specifications are located at each site can also be considered a replication problem.

14.3 ARCHITECTURAL ISSUES

As indicated in Chapter 4, one way to develop a distributed system is the client/server approach. Most, if not all, of the current object DBMSs are client/server systems. The design issues related to these systems are somewhat more complicated due to the characteristics of object models. Some of the concerns are listed below.

1. Since data and procedures are encapsulated as objects, the unit of communication between the clients and the server is an issue. The unit can be a page, an object, or a group of objects.

2. Closely related to the above issue is the design decision regarding the functions provided by the clients and the server. This is especially important since objects are not simply passive data, and it is necessary to consider the sites where object methods are executed.

3. In relational client/server systems, clients simply pass queries to the server, which executes them and returns the result tables to the client. This is referred to as *function shipping*. In object client/server DBMSs, this may not be the best approach, as the navigation of composite/complex object structures by the application program may dictate that data be moved to

the clients (called *data shipping systems*). Since data are shared by many clients, the management of client cache buffers for data consistency becomes a serious concern. Client cache buffer management is closely related to concurrency control, since data that is cached to clients may be shared by multiple clients, and this has to be controlled. Most commercial object DBMSs use locking for concurrency control, so a fundamental architectural issue is the placement of locks, and whether or not the locks are cached to clients.

4. Since objects may be composite or complex, there may be possibilities for prefetching component objects when an object is requested. Relational client/server systems do not usually prefetch data from the server, but this may be a valid alternative in the case of object DBMSs.

These considerations require revisiting some of the issues common to all DBMSs, along with several new ones. We will consider these issues in three sections: those directly related to architectural design—architectural alternatives, buffer management, and cache consistency—are discussed in this section; those related to object management—object identifier management, pointer swizzling, and object migration—are discussed in Section 14.4; and storage management related issues—object clustering and garbage collection—are considered in Section 14.5.

14.3.1 Alternative Client/Server Architectures

Two main types, of client/server architectures have been proposed: object servers and page servers. The distinction is partly based on the granularity of data that is shipped between the clients and the servers, and partly on the functionality provided to the clients and servers.

The first alternative is that clients request "objects" from the server, which retrieves them from the database and returns them to the requesting client. These systems are called *object servers* (Figure 14.1). In object servers, the server undertakes most of the DBMS services, with the client providing basically an execution environment for the applications, as well as some level of object management functionality (which will be discussed in Section 14.4). The object management layer is duplicated at both the client and the server in order to allow both to perform object functions. Object manager serves a number of functions. First and foremost, it provides a context for method execution. The replication of the object manager in both the server and the client enables methods to be executed at both the server and the clients. Executing methods in the client may invoke the execution of other methods, which may not have been shipped to the server with the object. The optimization of method executions of this type is an important research problem. Object manager also deals with the implementation of the object identifier (logical, physical, or virtual) and the deletion of objects (either explicit deletion or garbage collection). At the server, it also provides support for object clustering and access methods. Finally, the object managers at the client and the

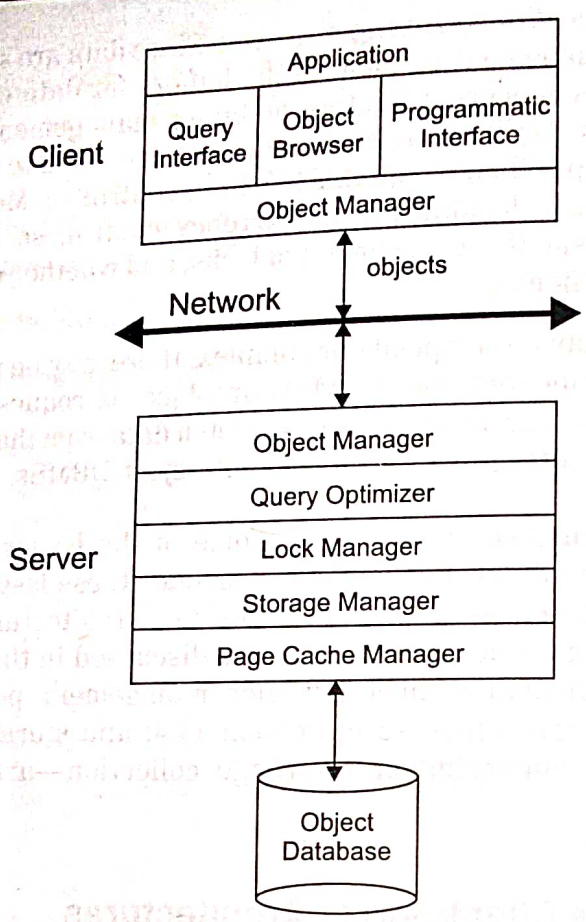

Figure 14.1. Object Server Architecture

server implement an object cache (in addition to the page cache at the server). Objects are cached at the client to improve system performance by localizing accesses. The client goes to the server only if the needed objects are not in its cache. The optimization of user queries and the synchronization of user transactions are all performed in the server, with the client receiving the resulting objects.

It is not necessary for servers in these architectures to send individual objects to the clients; if it is appropriate, they can send groups of objects. If the clients do not send any prefetching hints [Gerlhof and Kemper, 1994] then the groups correspond to contiguous space on a disk page. Otherwise, the groups can contain objects from different pages. Depending upon the group hit rate, the clients can dynamically either increase or decrease the group size [Liskov et al., 1996]. In these systems, one complication needs to be dealt with: clients return updated objects to clients. These objects have to be installed onto their corresponding data pages (called the *home page*). If the corresponding data page does not exist in the server buffer (such as, for example, if the server has already flushed it out), the server must perform an *installation read* to reload the home page for this object.

An alternative organization is a *page server* client/server architecture, in which the unit of transfer between the servers and the clients is a physical unit of data, such as a page or segment, rather than an object (Figure 14.2). Page server architectures split the object processing services between the clients and the servers. In fact, the servers do not deal with objects anymore, acting instead as "value-added" storage managers.

Early performance studies (e.g., [DeWitt et al., 1990]) favored page server architectures over object server architectures. In fact, these results have influenced an entire generation of research into the optimal design of page server-based object DBMSs. However, these results were not conclusive, since they indicated that page server architectures are better when there is a match between a data clustering pattern[4] and the users' access pattern, and that object server architectures are better when the users' data access pattern is not the same as the clustering pattern. These earlier studies were further limited in their consideration of only

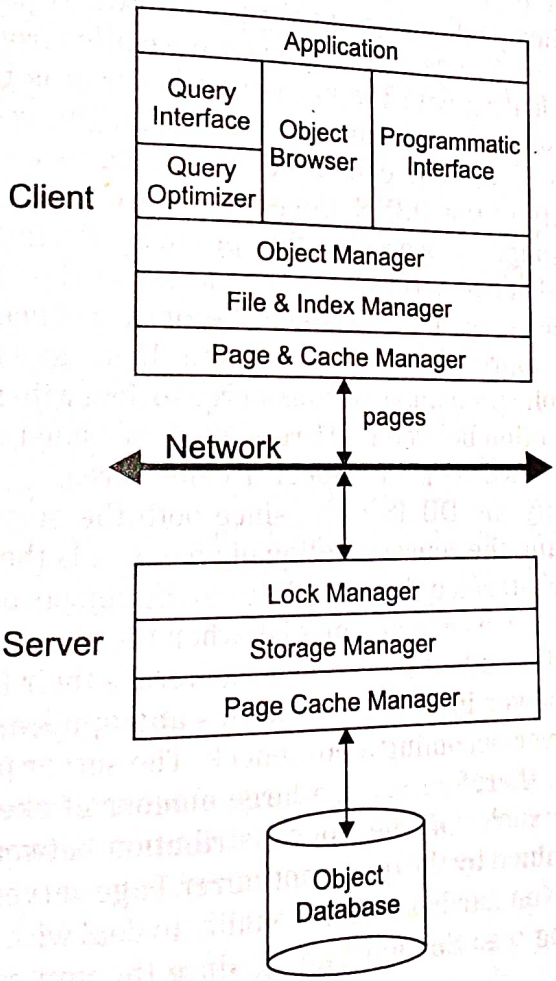

Figure 14.2. Page Server Architecture

[4]Clustering is an issue we will discuss later in this chapter. Briefly, it refers to how objects are placed on physical disk pages. Because of composite and complex objects, this becomes an important issue in object DBMSs.

single client/single server and multiple client/single server environments. There is clearly a need for further study in this area before a final judgment may be reached.

Intuitively, there should be significant performance advantages in having the server understand the "object" concept. One is that the server can apply locking and logging functions to the objects, enabling more clients to access the same page. Of course, this is relevant for small objects less than a page in size.

The second advantage is the potential for savings in the amount of data transmitted to the clients by filtering them at the server, which is possible if the server can perform some of the operations. This is indeed what the relational client/server syjitems do where the server is responsible for optimizing and executing the entire SQL query passed to it from a client. The situation is not as straightforward in object DBMSs, however, since the applications mix query access with object-by-object navigation. It is generally not a good idea to perform navigation at the server, since doing so would involve continuous interaction between the application and the server, resulting in a remote procedure call (RFC) for each object. In fact, the earlier studies were preferential towards page servers, since they mainly considered workloads involving heavy navigation from object to object.

One possibility of dealing with the navigation problem is to ship the user's application code to the server and execute it there as well. This requires significant care, however, since the user code cannot be considered safe and may threaten the safety and reliability of the DBMS. Some systems (e.g., Thor [Liskov et al., 1996]) use a safe language to overcome this problem. Furthermore, since the execution is now divided between the client and the server, data reside in both the server and the client cache, and its consistency becomes a concern. Nevertheless, the "function shipping" approach involving both the clients and the servers in the execution of a query/application must be considered to deal with mixed workloads. The distribution of execution between different machines must also be accommodated as systems move towards peer-to-peer architectures.

Page servers simplify the DBMS code, since both the server and the client maintain page caches, and the representation of an object is the same all the way from the disk to the user interface. Thus, updates to the objects occur only in client caches and these updates are reflected on disk when the page is flushed from the client to the server. Another advantage of page servers is their full exploitation of the client workstation power in executing queries and applications. Thus, there is less chance of the server becoming a bottleneck. The server performs a limited set of functions and can therefore serve a large number of clients. It is possible to design these systems such that the work distribution between the server and the clients can be determined by the query optimizer. Page servers can also exploit operating systems and even hardware functionality to deal with certain problems, such as pointer swizzling (see Section 14.4.2), since the unit of operation is uniformly a page.

Clearly, both of these architectures have important advantages and limitations. Unfortunately, the existing performance studies do not establish clear tradeoffs, even though they provide interesting insights.

Client Buffer Management

The clients can manage either a page buffer, an object buffer, or a dual (i.e., page/object) buffer. If clients have a page buffer, then entire pages are read or written from the server every time a page fault occurs or a page is flushed. Object buffers can read/write individual objects and allow the applications object-by-object access.

Object buffers manage access at a finer granularity and, therefore, can achieve higher levels of concurrency. However, they may experience buffer fragmentation, as the buffer may not be able to accommodate an integral multiple of objects, thereby leaving some unused space. A page buffer does not encounter this problem, but if the data clustering on the disk does not match the application data access pattern, then the pages contain a great deal of unaccessed objects that use up valuable client buffer space. In these situations, buffer utilization of a page buffer will be lower than the buffer utilization of an object buffer.

To realize the benefits of both the page and the object buffers, dual page/object buffers have been proposed [Kemper and Kossman, 1994], [Castro et al, 1997]. In a dual buffer system, the client loads pages into the page buffer. However, when the client flushes out a page, it retains the useful objects from the page by copying the objects into the object buffer. Therefore, the client buffer manager tries to retain well-clustered pages and isolated objects from non-well-clustered pages. The client buffer managers retain the pages and objects across the transaction boundaries (commonly referred to as *inter-transaction caching*). If the clients use a log-based recovery mechanism (see Chapter 12), they also manage an in-memory log buffer in addition to the data buffer. Whereas the data buffers are managed using a variation of the least recently used (LRU) policy, the log buffer typically uses a first-in/first-out buffer replacement policy. As in centralized DBMS buffer management, it is important to decide whether all client transactions at a station should share the cache, or whether each transaction should maintain its own private cache. The recent trend is for systems to have both shared and private buffers [Carey et al., 1994], [Biliris and Panagos, 1995].

Server Buffer Management

The server buffer management issues do not change in object client/server systems, since the servers usually manage a page buffer. We nevertheless discuss the issues here briefly in the interest of completeness. The pages from the page buffer are, in turn, sent to the clients to satisfy their data requests. A grouped object-server constructs its object groups by copying the necessary objects from the relevant server buffer pages, and sends the object group to the clients. In addition to the page level buffer, the servers can also maintain a modified object buffer (MOB) [Ghemawat, 1995]. A MOB stores objects that have been updated and returned by the clients. These updated objects have to be installed onto their corresponding data pages, which may require installation reads as described earlier. Finally, the modified page has to be written back to the disk. A MOB allows the server to

amortize its disk I/O costs by batching the installation read and installation write operations.

In a client/server system, since the clients typically absorb most of the data requests (i.e., the system has a high cache hit rate), the server buffer usually behaves more as a staging buffer than a cache. This, in turn, has an impact on the selection of server buffer replacement policies. Since it is desirable to minimize the duplication of data in the client and the server buffers, the *LRU with hate hints* buffer replacement policy can be used by the server [Franklin et al., 1992b]. The server marks the pages that also exist in client caches as *hated*. These pages are evicted first from the server buffer, and then the standard LRU buffer replacement policy is used for the remaining pages.

14.3.2 Cache Consistency

Cache consistency is a problem in any data shipping system that moves data to the clients. So the general framework of the issues discussed here also arise in relational client/server systems. However, the problems arise in unique ways in object DBMSs.

The study of DBMS cache consistency is very tightly coupled with the study of concurrency control, since cached data can be concurrently accessed by multiple clients, and locks can also be cached along with data at the clients. The DBMS cache consistency algorithms can be classified as avoidance-based or detection-based [Franklin et al., 1997]. *Avoidance-based algorithms* prevent the access of stale cache data[5] by ensuring that clients cannot update an object if it is being read by other clients. So they ensure that stale data never exists in client caches. *Detection-based algorithms* allow access of stale cache data, because clients can update objects that are being read by other clients. However, the detection-based algorithms perform a validation step at commit time to satisfy data consistency requirements.

Avoidance-based and detection-based algorithms can, in turn, be classified as *synchronous*, *asynchronous* or *deferred*, depending upon when they inform the server that a write operation is being performed. In synchronous algorithms, the client sends a lock escalation message at the time it wants to perform a write operation, and it blocks until the server responds. In asynchronous algorithms, the client sends a lock escalation message at the time of its write operation, but does not block waiting for a server response (it optimistically continues). In deferred algorithms, the client optimistically defers informing the server about its write operation until commit time. In deferred mode, the clients group all their lock escalation requests and send them together to the server at commit time. Thus, communication overhead is lower in a deferred cache consistency scheme, in comparison to synchronous and asynchronous algorithms.

The above classification results in a design space of possible algorithms covering six alternatives. Many performance studies have been conducted to assess the

[5] An object in a client cache is considered to be *stale* if that object has already been updated and committed into the database by a different client.

strengths and weaknesses of the various algorithms. In general, for data-caching systems, inter-transaction caching of data and locks is accepted as a performance enhancing optimization [Wilkinson and Neimat, 1990], [Franklin and Carey, 1994], because this reduces the number of times a client has to communicate with the server. On the other hand, for most user workloads, invalidation of remote cache copies during updates is preferred over propagation of updated values to the remote client sites [Franklin and Carey, 1994]. Hybrid algorithms that dynamically perform either invalidation or update propagation have been proposed [Franklin and Carey, 1994]. Furthermore, the ability to switch between page and object level locks is generally considered to be better than strictly dealing with page level locks [Carey et al., 1994] because it increases the level of concurrency.

Below, we will discuss each of the alternatives in the design space and comment on their performance characteristics.

- **Avoidance-based synchronous:** Callback-Read Locking (CBL) is the most common synchronous avoidance-based cache consistency algorithm [Franklin and Carey, 1994]. In this algorithm, the clients retain read locks across transactions, but they relinquish write locks at the end of the transaction. The clients send lock requests to the server and they block until the server responds. If the client requests a write lock on a page that is cached at other clients, the server issues callback messages requesting that the remote clients relinquish their read locks on the page. Callback-Read ensures a low abort rate and generally outperforms deferred avoidance-based, synchronous detection-based, and asynchronous detection-based algorithms.

- **Avoidance-based asynchronous:** Asynchronous avoidance-based cache consistency algorithms (AACC) [Özsu et al., 1998] do not have the message blocking overhead present in synchronous algorithms. Clients send lock escalation messages to the server and continue application processing. Normally, optimistic approaches such as this face high abort rates, but it is reduced in avoidance-based algorithms by immediate server actions to invalidate stale cache objects at remote clients as soon as the system becomes aware of the update. Thus, asynchronous algorithms experience lower deadlock abort rates than deferred avoidance-based algorithms, which are discussed next.

- **Avoidance-based deferred:** Optimistic Two-Phase Locking (O2PL) family of cache consistency are deferred avoidance-based algorithms [Franklin and Carey, 1994]. In these algorithms, the clients batch their lock escalation requests and send them to the server at commit time. The server blocks the updating client if other clients are reading the updated objects. As the data contention level increases, O2PL algorithms are susceptible to higher deadlock abort rates than CBL algorithms.

- **Detection-based synchronous:** Caching Two-Phase Locking (C2PL) is a synchronous detection-based cache consistency algorithm [Carey et al.,

1991]. In this algorithm, clients contact the server whenever they access a page in their cache to ensure that the page is not stale or being written to by other clients. C2PL's performance is generally worse than CBL and O2PL algorithms, since it does not cache read locks across transactions.

- **Detection-based asynchronous:** No-Wait Locking (NWL) with Notification is an asynchronous detection-based algorithm [Wang and Rowe, 1991]. In this algorithm, the clients send lock escalation requests to the server, but optimistically assume that their requests will be successful. After a client transaction commits, the server propagates the updated pages to all the other clients that have also cached the affected pages. It has been shown that CBL outperforms the NWL algorithm.

- **Detection-based deferred:** Adaptive Optimistic Concurrency Control (AOCC) is a deferred detection-based algorithm. It has been shown [Adya et al., 1995] that AOCC can outperform callback locking algorithms even while encountering a higher abort rate if the client transaction state (data and logs) completely fits into the client cache, and all application processing is strictly performed at the clients (purely data-shipping architecture). Since AOCC uses deferred messages, its messaging overhead is less than CBL. Furthermore, in a purely data-shipping client/server environment, the impact of an aborting client on the performance of other clients is quite minimal. These factors contribute to AOCC's superior performance.

14.4 OBJECT MANAGEMENT

The exact nature of object management functionality is open to discussion. As indicated in Section 14.3.1, this includes tasks such as object identifier management, pointer swizzling, object migration, deletion of objects, method execution, and some storage management tasks at the server. In this section we will discuss some of these problems; those related to storage management are discussed in the next section.

14.4.1 Object Identifier Management

As indicated in Section 14.1, object identifiers (OIDs) are system-generated and used to uniquely identify every object (transient or persistent, system-created or user-created) in the system. Implementing the identity of persistent objects generally differs from implementing transient objects, since only the former must provide global uniqueness. In particular, transient object identity can be implemented more efficiently.

The implementation of persistent object identity has two common solutions, based on either physical or logical identifiers, with their respective advantages and shortcomings. The physical identifier (POID) approach equates the OID with the physical address of the corresponding object. The address can be a disk page address and an offset from the base address in the page. The advantage is that

the object can be obtained directly from the OID. The drawback is that all parent objects and indexes must be updated whenever an object is moved to a different page.

The logical identifier (LOID) approach consists of allocating a systemwide unique OID (i.e., a surrogate) per object. Since OIDs are invariant, there is no overhead due to object movement. This is achieved by an OID table associating each OID with the physical object address at the expense of one table look-up per object access. To avoid the overhead of OIDs for small objects that are not referentially shared, both approaches can consider the object value as their identifier. The earlier hierarchical and network database systems used the physical identifier approach. Object-oriented database systems tend to prefer the logical identifier approach, which better supports dynamic environments.

Implementing transient object identity involves the techniques used in programming languages. As for persistent object identity, identifiers can be physical or logical. The physical identifier can be the real or virtual address of the object, depending on whether virtual memory is provided. The physical identifier approach is the most efficient, but requires that objects do not move. Theological identifier approach, promoted by object-oriented programming, treats objects uniformly through an indirection table local to the program execution. This table associates a logical identifier, called an *object oriented pointer* (OOP) in Smalltalk, to the physical identifier of the object. Object movement is provided at the expense of one table look-up per object access.

The dilemma for an object manager is a trade-off between generality and efficiency. The general support of the object model incurs a certain overhead. For example, object identifiers for small objects can make the OID table quite large. By limiting the support of the object model—for example, by not providing object snaring directly, and by relying on higher levels of system (e.g., the compiler of the database language) for that support, more efficiency may be gained. Object identifier management is closely related to object storage techniques, which we will discuss in Section 14.5.

In distributed object DBMSs, it may be more appropriate to use LOIDs, since operations such as reclustering, migration, replication and fragmentation occur frequently. The use of LOIDs raises the following distribution related issues:

- **LOID Generation:** LOIDs must be unique within the scope of the entire distributed domain. It is relatively easy to ensure uniqueness if the LOIDs are generated at a central site. However, a centralized LOID generation scheme is not desirable because of the network latency overhead and the load on the LOID generation site. In multi-server environments, each server site generates LOIDs for the objects stored at that site. The uniqueness of the LOID is ensured by incorporating the server identifier as part of the LOID. Therefore, the LOID consists of both a server identifier part and a sequence number. The sequence number is the logical representation of the disk location of the object. The sequence numbers are unique within a particular server, and are usually not re-used to prevent existing references to the deleted object from pointing to the new object which assumes

the same sequence number. During object access time, if the server identifier portion of the LOID is not directly used for object location identification, the object identifier functions as a pure LOID. However, if the server identifier portion of the LOID is used, the LOID functions as a pseudo-LOID.

- **LOID Mapping Location and Data Structures:** The location of the LOID-to-POID mapping information is important. If pure LOIDs are used, and if a client can be directly connected to multiple servers simultaneously, then the LOID-to-POID mapping information must be present at the client. If pseudo-LOIDs are used, the mapping information needs to be present only at the server. The presence of the mapping information at the client is not desirable, because this solution is not scalable (i.e., the mapping information has to be updated at all the clients which might access the object).

  The LOID-to-POID mapping information is usually stored in hash tables or in B+ trees. There are advantages and disadvantages to both [Eicker et al., 1995]. Hash tables provide fast access, but are not scalable as the database size increases. B+ trees are scalable, but have a logarithmic access time, and require complex concurrency control and recovery strategies.

14.4.2 Pointer Swizzling

In object DBMSs, one can navigate from one object to another using *path expressions* that involve attributes with object-based values (e.g., if `c` is of type `Car`, then `c.engine.manufacturer.name` is a path expression[6]). These are basically pointers. Usually on disk, object identifiers are used to represent these pointers. However, in memory, it is desirable to use in-memory pointers for navigating from one object to another. The process of converting a disk version of the pointer to an in-memory version of a pointer is known as "pointer-swizzling". Hardware-based and software-based schemes are two types of pointer-swizzling mechanisms [White and DeWitt, 1994]. In hardware-based schemes, the operating system's page-fault mechanism is used; when a page is brought into memory, all the pointers in it are swizzled, and they point to reserved virtual memory frames. The data pages corresponding to these reserved virtual frames are only loaded into memory when an access is made to these pages. The page access, in turn, generates an operating system page-fault, which must be trapped and processed, In software-based schemes, an object table is used for pointer-swizzling purposes. That is, a pointer is swizzled to point to a location in the object table. There are eager and lazy variations to the software-based schemes, depending upon when exactly the pointer is swizzled. Therefore, every object access has a level of indirection associated with it. The advantage of the hard ware-based scheme is that it leads to

[6] We assume that `Engine` type is defined with at least one attribute, `manufacturer`, whose domain is the extent of type `Manufacturer`. `Manufacturer` type has an attribute called `name`.

better performance when repeatedly traversing a particular object hierarchy, due to the absence of a level of indirection for each object access. However, in bad clustering situations where only a few objects per page are accessed, the high overhead of the page-fault handling mechanism makes hardware-based schemes unattractive. Hardware-based schemes also do not prevent client applications from accessing deleted objects on a page. Moreover, in badly clustered situations, hardware-based schemes can exhaust the virtual memory address space, because page frames are aggressively reserved regardless of whether the objects in the page are actually accessed. Finally, since the hardware-based scheme is implicitly page-oriented, it is difficult to provide object-level concurrency control, buffer management, data transfer and recovery features. In many cases, it is desirable to manipulate data at the object level rather than the page level.

14.4.3 Object Migration

One aspect of distributed systems is that objects move, from time to time, between sites. This raises a number of issues. First is the unit of migration. In systems where the state is separated from the methods, it is possible to consider moving the object's state without moving the methods. The counterpart of this scenario in purely behavioral systems is the fragmentation of an object according to its behaviors. In either case, the application of methods to an object requires the invocation of remote procedures. This issue was discussed above under object distribution. Even if individual objects are units of migration [Dollimore et al., 1994], their relocation may move them away from their type specifications and one has to decide whether types are duplicated at every site where instances reside or whether the types are accessed remotely when behaviors or methods are applied to objects. Three alternatives can be considered for the migration of classes (types):

1. the source code is moved and recompiled at the destination,
2. the compiled version of a class is migrated just like any other object, or
3. the source code of the class definition is moved, but not its compiled operations, for which a lazy migration strategy is used.

Another issue is that the movements of the objects must be tracked so that they can be found in their new locations. A common way of tracking objects is to leave *surrogates* [Hwang, 1987], [Liskov et al., 1994], or *proxy objects* [Dickman, 1994]. These are place-holder objects left at the previous site of the object, pointing to its new location. Accesses to the proxy objects are directed transparently by the system to the objects themselves at the new sites. The migration of objects can be accomplished based on their current state [Dollimore et al., 1994]. Objects can be in one of four states:

1. **Ready:** Ready objects are not currently invoked, or have not received a message, but are ready to be invoked to receive a message.

2. **Active:** Active objects are currently involved in an activity in response to an invocation or a message.
3. **Waiting:** Waiting objects have invoked (or have sent a message to) another object and are waiting for a response.
4. **Suspended:** Suspended objects are temporarily unavailable for invocation.

Objects in active or waiting state are not allowed to migrate, since the activity they are currently involved in would be broken: The migration involves two steps:

1. shipping the object from the source to the destination, and
2. creating a proxy at the source, replacing the original object.

Two related issues must also be addressed here. One relates to the maintenance of the system directory. As objects move, the system directory must be updated to reflect the new location. This may be done lazily, whenever a surrogate or proxy object redirects an invocation, rather than eagerly, at the time of the movement. The second issue is that, in a highly dynamic environment where objects move frequently, the surrogate or proxy chains may become quite long. It is useful for the system to transparently compact these chains from time to time. However, the result of compaction must be reflected in the directory, and it may not be possible to accomplish that lazily.

Another important migration issue arises with respect to the movement of composite objects. The shipping of a composite object may involve shipping other objects referenced by the composite object. An alternative method of dealing with this is a method called *object assembly*, which we will consider under query processing in Section 14.6.3.

14.5 DISTRIBUTED OBJECT STORAGE

Among the many issues related to object storage, two are particularly relevant in a distributed system: object clustering and distributed garbage collection. Composite and complex objects provide opportunities, as we mentioned earlier, for clustering data on disk such that the I/O cost of retrieving them is reduced. Garbage collection is a problem that arises in object databases, since they allow reference-based sharing. Thus, object deletion and subsequent storage reclamation requires special care.

Object Clustering

An object model is essentially conceptual, and should provide high physical data independence to increase programmer productivity. The mapping of this conceptual model to a physical storage is a classical database problem. As indicated in Section 14.1, in the case of object DBMSs, two kinds of relationships exist between

types: subtyping and composition. By providing a good approximation of object access, these relationships are essential to guide the physical clustering of persistent objects. Object clustering refers to the grouping of objects in physical containers (i.e., disk extents) according to common properties, such as the same value of an attribute or sub-objects of the same object. Thus, fast access to clustered objects can be obtained.

Object clustering is difficult for two reasons. First, it is not orthogonal to object identity implementation (i.e, logical vs. physical OID). Logical OIDs incur more overhead (an indirection table), but enable vertical partitioning of classes. Physical OIDs yield more efficient direct object access, but require each object to contain all inherited attributes. Second, the clustering of complex objects along the composition relationship is more involved because of object sharing (objects with multiple parents).

Given a class graph, there are three basic storage models for object clustering [Valduriez et al., 1986].

1. The *decomposition storage model* (DSM) partitions each object class in binary relations (OID, attribute) and therefore relies on logical OID. The advantage of DSM is simplicity.

2. The *normalized storage model* (NSM) stores each class as a separate relation. It can be used with logical or physical OID. However, only logical OID allows the vertical partitioning of objects along the inheritance relationship [Kim et al., 1987].

3. The *direct storage model* enables multi-class clustering of complex objects based on the composition relationship. This model generalizes the techniques of hierarchical and network databases, and works best with physical OID [Benzaken and Delobel, 1991]. It can capture object access locality and is therefore potentially superior for well-known access patterns. The major difficulty, however, is to recluster an object whose parent has been deleted.

In a distributed system, both DSM and NSM are straightforward, using horizontal partitioning. Goblin [Kersten et al., 1993] implements DSM as a basis for a distributed object DBMS with large main memory. DSM provides flexibility, and its performance disadvantage is compensated by the use of large main memory and caching. Eos [Gruber and Amsaleg, 1993] implements the direct storage model in a distributed single-level store architecture, where each object has a physical, system-wide OID. The Eos grouping mechanism is based on the concept of most relevant composition links and solves the problem of multiparent shared objects. When an object moves to a different node, it gets a new OID. To avoid the indirection of forwarders, references to the object are subsequently changed as part of the garbage collection process without any overhead. The grouping mechanism is dynamic to achieve load balancing and cope with the evolutions of the object graph.

Distributed Garbage Collection

An advantage of object-based systems is that objects can refer to other objects using object identity. As programs modify objects and remove references, a persistent object may become unreachable from the persistent roots of the system when there is no more reference to it. Such an object is "garbage" and should be de-allocated by the garbage collector. In relational DBMSs, there is no need for automatic garbage collection, since object references are supported by join values. However, cascading updates as specified by referential integrity constraints are a simple form of "manual" garbage collection. In more general operating system or programming language contexts, manual garbage collection is typically error-prone. Therefore, the generality of distributed object-based systems calls for automatic distributed garbage collection.

The basic garbage collection algorithms can be categorized as reference counting or tracing-based. In a reference counting system, each object has an associated count of the references to it. Each time a program creates an additional reference that points to an object, the object's count is incremented. When an existing reference to an object is destroyed, the corresponding count is decremented. The memory occupied by an object can be reclaimed when the object's count drops to zero (at which time, the object is garbage).

Tracing-based collectors are divided into *mark and sweep* and *copy-based* algorithms. *Mark and sweep* collectors are two-phase algorithms. The first phase, called the "mark" phase, starts from the root and marks every reachable object (for example, by setting a bit associated to each object). This mark is also called a "color", and the collector is said to color the objects it reaches. The mark bit can be embedded in the objects themselves or in *color maps* that record, for every memory page, the colors of the objects stored in that page. Once all live objects are marked, the memory is examined and unmarked objects are reclaimed. This is the "sweep" phase.

Copy-based collectors divide memory into two disjoint areas called *from-space* and *to-space*. Programs manipulate from-space objects, while the to-space is left empty. Instead of marking and sweeping, copying collectors copy (usually in a depth first manner) the from-space objects reachable from the root into the to-space. Once all live objects have been copied, the collection is over, the contents of the from-space are discarded, and the roles of from- and to-spaces are exchanged. The copying process copies objects linearly in the to-space, which compacts memory.

The basic implementations of mark and sweep and copy-based algorithms are "stop-the-world"; i.e., user programs are suspended during the whole collection cycle. For many applications, however, stop-the-world algorithms cannot be used because of their disruptive behavior. Preserving the response time of user applications requires the use of incremental techniques. Incremental collectors must address problems raised by concurrency. The main difficulty with incremental garbage collection is that, while the collector is tracing the object graph, program activity may change other parts of the object graph. In some cases, the collector may miss tracing some reachable objects, and thus may erroneously reclaim them.

Designing a garbage collection algorithm for object DBMSs is difficult. These systems have several features that pose additional problems for incremental garbage collection, beyond those typically addressed by solutions for non-persistent systems. These problems include the ones raised by the resilience to system failures and the semantics of transactions, and, in particular, by the rollbacks of partially completed transactions, by traditional client-server performance optimizations (such as client caching and flexible management of client buffers), and by the huge volume of data to analyze in order to detect garbage objects.

Distributed garbage collectors rely either on distributed reference counting or distributed tracing. Distributed reference counting is problematic for two reasons. First, reference counting cannot collect unreachable cycles of garbage objects (i.e., mutually-referential garbage objects). Second, reference counting is defeated by common message failures; that is, if messages are not delivered reliably in their causal order, then maintaining the reference counting invariant (i.e., equality of the count with the actual number of references) is problematic.

14.6 OBJECT QUERY PROCESSING

Relational DBMSs have benefitted from the early definition of a precise and formal query model and a set of universally-accepted algebraic primitives. This has not been the case with object DBMSs. The first-generation object DBMSs did not have a declarative query language, and some still do not. It was commonly believed that the application domains that these systems target did not need querying capabilities. This belief no longer holds, and declarative query capability is now accepted as a fundamental feature of object DBMSs [Atkinson et al., 1989], [Stonebraker et al., 1990a].

There is a close relationship between query optimization techniques and the query model and query language. For example, a functional query language lends itself to functional optimization, which is quite different from the algebraic, cost-based optimization techniques used in both relational systems and a number of object-oriented systems. The query model, in turn, is based on the data (or object) model, since the latter defines the access primitives used by the query model. These primitives at least partially determine the power of the query model. The object model issues were discussed in the previous section; we will not dwell long on query model issues here since the focus of this section is on query processing and optimization. For the remainder, we will assume that the user query language is OQL [Cattell, 1997], which is an object-oriented version of the SQL language upon which we have relied heavily in previous chapters. OQL was developed by a consortium of object DBMS vendors (known as the Object Database Group or ODBG), so it serves as a de facto industry standard. Some systems (e.g., O_2 [Deux et al., 1991]) already provide OQL interfaces. As we did earlier, we will take liberties with the language syntax.

Almost all object query processors that have been proposed to date use the optimization techniques that have been developed for relational systems. However, there are a number of issues that make query processing and optimization more

difficult in object DBMSs. The more important issues are the following [Özsu and Blakeley, 1994]:

1. Relational query languages operate on very simple type systems consisting of a single type: relation. The closure property of relational languages implies that each relational operator takes one or two relations as operands and generates a relation as a result. In contrast, object systems have richer type systems. The results of object algebra operators are usually sets of objects (or collections), which may be of different types. If the object languages are closed under the algebra operators, these heterogeneous sets of objects can be operands to other operators. This requires the development of elaborate type inferencing schemes to determine which methods can be applied to all the objects in such a set. Furthermore, object algebras often operate on semantically different collection types (e.g., set, bag, list), which imposes additional requirements on the type inferencing schemes to determine the type of the results of operations on collections of different types.

2. Relational query optimization depends on knowledge of the physical storage of data (access paths) which is readily available to the query optimizer. The encapsulation of methods with the data upon which they operate in object DBMSs raises at least two important issues. First, determining (or estimating) the cost of executing methods is considerably more difficult than calculating or estimating the cost of accessing an attribute according to an access path. In fact, optimizers have to worry about optimizing method execution, which is not an easy problem because methods may be written using a general-purpose programming language. Second, encapsulation raises issues related to the accessibility of storage information by the query optimizer. Some systems overcome this difficulty by treating the query optimizer as a special application that can break encapsulation and access information directly [Cluet and Delobel, 1992]. Others propose a mechanism whereby objects "reveal" their costs as part of their interface [Graefe and Maier, 1988].

3. Objects can (and usually do) have complex structures whereby the state of an object references another object. Accessing such complex objects involves *path expressions*. The optimization of path expressions is a difficult and central issue in object query languages. We discuss this issue in some detail in this chapter. Furthermore, objects belong to types related through inheritance hierarchies. Optimizing the access to objects through their inheritance hierarchies is also a problem that distinguishes object-oriented from relational query processing.

4. As mentioned earlier, one of the facts of life in object DBMSs is the lack of a universally-accepted object model definition. Even though there is some convergence in the set of basic features that must be supported by any object model (such as object identity, encapsulation of state and behavior,

type inheritance, and typed collections), how these features are supported differs among models and systems. As a result, the numerous projects that experiment with object query optimizers follow quite different paths and are, to a certain degree, incompatible, making it quite difficult to amortise on the experiences of others. As this diversity of approaches is likely to prevail for some time, extensible approaches to query optimization that allow experimentation with new ideas as they evolve are important in object query processing. We provide an overview of the various extensibility approaches.

Object query processing and optimization has been the subject of significant research activity. Unfortunately, most of this work has not been extended to distributed object systems. Therefore, in the remainder of this chapter, we will restrict ourselves to a summary of the important issues: object query processing architectures (Section 14.6.1), object query optimization (Section 14.6.2), and query execution strategies (Section 14.6.3).

14.6.1 Object Query Processor Architectures

As indicated in Chapter 7, query optimization can be modeled as an optimization problem whose solution is the choice, based on a *cost function*, of the "optimum" *state*, which corresponds to an algebraic query, in a *state space* (also called *search space*) that represents a family of equivalent algebraic queries. Query processors differ, architecturally, according to how they model these components.

Many existing object DBMS optimizers are either implemented as part of the object manager on top of a storage system, or as client modules in a client/server architecture. In most cases, the above-mentioned components are "hardwired" into the query optimizer. Given that extensibility is a major goal of object DBMSs, one would hope to develop an extensible optimizer that accommodates different search strategies, algebra specifications (with their different transformation rules), and cost functions. Rule-based query optimizers [Freytag, 1987], [Graefe and DeWitt, 1987] provide some amount of extensibility by allowing the definition of new transformation rules. However, they do not allow extensibility in other dimensions. In this section we will discuss some new, promising proposals for extensibility in object DBMSs.

The Open OODB project [Wells et al., 1992] at Texas Instruments concentrates on the definition of an open architectural framework for object DBMSs and describes the design space for these systems. The query module is an example of intra-module extensibility in Open OODB. The query optimizer [Blakeley et al., 1993], built using the Volcano optimizer generator [Graefe and McKenna, 1993], is extensible with respect to algebraic operators, logical transformation rules, execution algorithms, implementation rules (i.e., logical operator-to-execution algorithm mappings), cost estimation functions, and physical property enforcement functions (e.g., presence of objects in memory). The separation between the user query language parsing structures and the operator graph on which the

optimizer operates allows the replacement of the user language or optimizer. The separation between algebraic operators and execution algorithms allows exploration with alternative methods for implementing algebraic operators. Code generation is also a well-defined subcomponent of the query module, which facilitates porting the query module to work on top of other object DBMSs. The Open OODB query processor includes a query execution engine containing efficient implementations of scan, indexed scan, hybrid-hash join [Shapiro, 1986], and complex object assembly [Keller et al., 1991], which we discuss later.

14.6.2 Query Processing Issues

Query processing methodology in object DBMSs is similar to its relational counterpart, as discussed in Chapters 7–9, but with differences in details as a result of the object model and query model characteristics. In this section we will consider these differences as they apply to algebraic optimization. We will also discuss a particular problem unique to object query models — namely, the execution of path expressions.

Algebraic Optimization

Search Space and Transformation Rules. As discussed in Chapter 8, a major advantage of algebraic optimization is that an algebraic query expression can be transformed using well-defined algebraic properties, such as transitivity, commutativity and distributivity. During the process, one eliminates plans with execution times that are worse than the previously-found minimum.

The transformation rules are very much dependent upon the specific object algebra, since they are defined individually for each object algebra and for their combinations. The lack of a standard object algebra definition is particularly troubling since the community cannot benefit from generalizations of numerous studies. The general considerations for the definition of transformation rules and the manipulation of query expressions is quite similar to relational systems, with one particularly important difference. Relational query expressions are defined on flat relations, whereas object queries are defined on classes (or collections or sets of objects) that have subtyping and composition relationships among them. It is, therefore, possible to use the semantics of these relationships in object query optimizers to achieve some additional transformations.

Consider, for example, three object algebra operators: [Straube and Özsu, 1990a] union (denoted $\cup$), intersection (denoted $\cap$) and parameterized *select* (denoted $P\sigma_F < Q_1 \ldots Q_k >$), where union and *intersection* have the usual set-theoretic semantics, and *select* selects objects from one set P using the sets of objects $Q_1 \ldots Q_k$ as parameters (in a sense, a generalized form of semijoin). The results of these operators are sets of objects as well. The following are some of the transformation rules that can be applied during optimization to get equivalent query expressions (for brevity, we use *QSet* to denote $Q_1 \ldots Q_k$; *RSet* is defined similarly):

$$(P\sigma_{F_1} < QSet >)\sigma_{F_2} < RSet > \Leftrightarrow (P\sigma_{F_2} < RSet >)\sigma_{F_1} < QSet >$$
$$(P \cup Q)\sigma_F < RSet > \Leftrightarrow (P\sigma_F < RSet >) \cup (Q\sigma_F < RSet >)$$
$$(P\sigma_{F_1} < QSet >)\sigma_{F_2} < RSet > \Leftrightarrow (P\sigma_{F_1} < QSet >) \cap (P\sigma_{F_2} < RSet >)$$

The first rule captures commutativity of `select`, while the second rule denotes that `select` distributes over union. The third rule is an identity which uses the fact that `select` merely restricts its input and returns a subset of its first argument[7].

The first two rules are quite general in that they represent equivalences inherited from set theory. The third is a special transformation rule for a specific object algebra operator defined with a specific semantics. All three, however, are syntactic in nature. Consider the following rules, on the other hand, where C_i denotes the set of objects in the extent of class c_i and C_j^* denotes the deep extent of class c_j (i.e., the set of objects in the extent of c_j, as well as in the extents of all those which are subclasses of c_j):

$$C_1 \cap C_2 = \phi \text{ if } c_1 \neq c_2$$
$$C_1 \cup C_2^* = C_2^* \text{ if } c_1 \text{ is a subclass of } c_2$$
$$(P\sigma_F < QSet >) \cap R \overset{c}{\Leftrightarrow} (P\sigma_F < QSet >) \cap (R\sigma_{F'} < QSet >) \overset{c}{\Leftrightarrow} P \cap (R\sigma_{F'} < QSet >)$$

These transformation rules are semantic in nature, since they depend on the object model and query model specifications. The first rule, for example, is true because the object model restricts each object to belong to only one class. The second rule holds because the query model permits retrieval of objects in the deep extent of the target class. Finally, the third rule relies on type consistency rules [Straube and Ozsu, 1990b] for its applicability, as well as a condition (denoted by the c over the $\Leftrightarrow$) that F' is identical to F, except, that each occurrence of p is replaced by r:

Since the idea of query transformation is well-known, we will not elaborate on the techniques. The above discussion only demonstrates the general idea and highlights the unique aspects that must be considered in object algebras.

Search Algorithm. As discussed in Chapters 8 and 9, exhaustive search algorithms enumerate the entire search space, applying a cost function to each equivalent expression to determine the least expensive one. An improvement is to use a dynamic programming approach, whereby new expressions are constructed bottom-up using the previously-determined optimal subexpressions [Lee et al., 1988], [Selinger et al., 1979]. The Volcano optimizer generator uses a top-down, dynamic programming approach to search with branch-and-bound pruning [Graefe and McKenna, 1993]. These are called *enumerative algorithms*.

The combinatorial nature of enumerative search algorithms is perhaps more important in object DBMSs than in relational ones. It has been argued that if the number of joins in a query exceeds ten, enumerative search strategies become infeasible [Ioannidis and Wong, 1987]. In applications such as decision support

[7] These rules make assumptions about the formulae (F_i), which we will not address in this chapter.

systems which object DBMSs are well-suited to support, it is quite common to find queries of this complexity. Furthermore, as we will address in Section 14.6.2, one method of executing path expressions is to represent them as explicit joins, and then use the well-known join algorithms to optimize them. If this is the case, the number of joins and other operations with join semantics in a query is quite likely to be higher than the empirical threshold of ten.

In these cases, *randomized search algorithms* (which we introduced in Chapters 8 and 9) have been suggested as alternatives to restrict the region of the search space being analyzed. Unfortunately, there has not been any study of randomized search algorithms within the context of object DBMSs. The general strategies are not likely to change, but the tuning of the parameters and the definition of the space of acceptable solutions should be expected to change. It is also interesting to note the surface similarity between randomized search algorithms and the regions approach proposed by Mitchell, et al [1993]. Further studies are required to establish the relationship more firmly. Furthermore, the distributed versions of these algorithms are not available, and their development remains a challenge.

Cost Function. As we have already seen, the arguments to cost functions are based on various information regarding the storage of the data. Typically, the optimizer considers the number of data items (cardinality), the size of each data item, its organization (e.g., whether there are indexes on it or not), etc. This information is readily available to the query optimizer in relational systems (through the system catalog), but may not be in object DBMSs. As indicated earlier, there is a controversy in the research community as to whether the query optimizer should be able to break the encapsulation of objects and look at the data structures used to implement them. If this is permitted, the cost functions can be specified similar to relational systems [Blakeley et al., 1993], [Cluet and Delobel, 1992], [Dogac et al., 1994], [Orenstein et al., 1992]. Otherwise, an alternative specification must be considered.

The cost function can be defined recursively based on the algebraic processing tree. If the internal structure of objects is not visible to the query optimizer, the cost of each node (representing an algebraic operation) has to be defined. One way to define it is to have objects "reveal" their costs as part of their interface [Graefe and Maier, 1988]. A similar approach is provided in the TIGUKAT project [Özsu et al., 1995b]. Since the algebraic operations are behaviors defined on type `Collection`, the nodes of the algebraic processing tree are behavior applications. There may be various functions that implement each behavior (representing different execution algorithms), in which case the behaviors "reveal" their costs as a function of (a) the execution algorithm and (b) the collection over which they operate. In both cases, a more abstract cost function for behaviors is specified at type definition time from which the query optimizer can calculate the cost of the entire processing tree. The definition of cost functions, especially in the approaches based on the objects revealing their costs, must be investigated further before satisfactory conclusions can be reached.

Parameterization. Compile-time query optimization is a static process in that the optimizer uses the database statistics at the time the query is compiled and optimized in selecting the optimal execution plan. This decision is independent of the execution-time statistics, such as the system load. Furthermore, it does not take into account the changes to the database statistics as a result of updates that may occur between the time the query is optimized and the time it is executed. This is especially a problem in production-type queries which are optimized once (with considerable overhead) and executed a large number of times. It may be an even more serious issue in object DBMSs which may be used as repositories for design prototypes (software or otherwise). These databases are by definition more volatile, resulting in significant changes to the database (which is why dynamic schema evolution is so important in object DBMSs). The query optimization strategy must be able to cope with these changes.

The issue can be handled in one of two ways. One alternative is to determine an optimization/re-optimization interval and re-optimize the query periodically. Even though this is a simple approach, it is based on a fixed time interval whose determination in general would be problematic. A slight variation may be to determine the re-optimization point based on the difference between the actual execution time and the estimated execution time. Consequently, the run-time system will be able to track the actual execution time, and whenever it deviates from the estimated time by more than a fixed threshold, the query will be re-optimized. Again, the determination of this threshold would be a concern, as well as the run-time overhead of tracking query execution.

Another alternative that has been researched [Graefe and Ward, 1989], [Ioannidis et al, 1992] and implemented in ObjectStore [Orenstein et al., 1992] is *parametric query optimization*, which is also called *dynamic plan selection*. In this case, the optimizer maintains multiple execution strategies at compile time and makes a final plan selection at run-time based on various system parameters and the current database statistics. If the optimizer does not have access to all of this data, algebraic optimization can ignore all physical execution characteristics, instead generating a set of "desirable" (however defined) equivalent query expressions which are handed over to the object manager. The object manager can then compare the alternatives (at run time) based on their execution characteristics. However, this approach also has the significant problem of potentially incurring high run-time overhead.

A problem with compile-time parametric optimization (and run-time resolution) is the potential exponential explosion of the dynamic plans as a function of both the complexity of the query and the number of optimization parameters unknown at compile time. This problem, along with the problems of error propagation and inaccuracy of selectivity and cost estimation methods, makes "run-time" query optimization an attractive alternative.

Path Expressions

Most object query languages allow queries whose predicates involve conditions on object access along reference chains. These reference chains are called *path*

expressions [Zaniolo, 1983] (sometimes also referred to as complex predicates or *implicit joins* [Kim, 1989]). The example path expresion `c.engine.manufacturer.name` that we used in Section 14.4.2 retrieves the value of the `name` attribute of the object that is the value of the `manufacturer` attribute of the object that is the value of the `engine` attribute of object c, which was defined to be of type `Car`. It is possible to form path expressions involving attributes as well as methods. Optimizing the computation of path expressions is a problem that has received substantial attention in object-query processing.

Path expressions allow a succinct, high-level notation for expressing navigation through the object composition (aggregation) graph, which enables the formulation of predicates on values deeply nested in the structure of an object. They provide a uniform mechanism for the formulation of queries that involve object composition and inherited member functions. Path expressions may be single-valued or set-valued, and may appear in a query as part of a predicate, a target to a query (when set-valued), or part of a projection list. A path expression is single-valued if every component of a path expression is single-valued; if at least one component is set-valued, then the whole path expression is set-valued. Techniques to traverse path expressions forward and backward are presented by Jenq et al. [1990].

The problem of optimizing path expressions spans the entire query-compilation process. During or after parsing of a user query, but before algebraic optimization, the query compiler must recognize which path expressions can potentially be optimized. This is typically achieved through *rewriting* techniques, which transform path expressions into equivalent logical algebra expressions [Cluet and Delobel, 1992]. Once path expressions are represented in algebraic form, the query optimizer explores the space of *equivalent algebraic* and execution plans, searching for one of minimal cost [Lanzelotte and Valduriez, 1991], [Blakeley et al., 1993). Finally, the optimal execution plan may involve algorithms to efficiently compute path expressions, including hash-join [Shapiro, 1986], complex-object assembly [Keller et al., 1991 or indexed scan through path indexes [Maier and Stein, 1986], [Valduriez, 1987 [Kemper and Moerkotte, 1990a], [Kemper and Moerkotte, 1990b].

Rewriting and Algebraic Optimization. Consider again the path expression `c.engine.manufacturer.name`. Assume every car instance has a reference to an `Engine` object, each engine has a reference to a `Manufacturer` object, and each manufacturer instance has a `name` field. Also, assume that `Engine` and `Manufacturer` types have a corresponding type extent. The first two links of the above path may involve the retrieval of engine and manufacturer objects from disk. The third path involves only a lookup of a field within a manufacturer object. Therefore, only the first two links present opportunities for query optimization in the computation of that path. An object-query compiler needs a mechanism to distinguish these links in a path representing possible optimizations. This is typically achieved through a *rewriting* phase.

One possibility is to use a type-based rewriting technique, as proposed by Cluet and Delobel [1992]. This approach "unifies" algebraic and type-based rewriting techniques, permits factorization of common subexpressions, and supports

heuristics to limit rewriting. Type information is exploited to decompose initial complex arguments of a query into a set of simpler operators, and to rewrite path expressions into joins. Lanzelotte and Valduriez [1991] present a similar attempt to optimizing path expressions within an algebraic framework using an operator called *implicit join*. Rules are defined to transform a series of implicit join operators into an indexed scan using a path index (see below) when it is available.

An alternative operator that has been proposed for optimizing path expressions is *materialize* (Mat) [Blakeley et al., 1993], which represents the computation of each inter-object reference (i.e., path link) explicitly. This enables a query optimizer to express the materialization of multiple components as a group using a single `Mat` operator, or individually using a Mat operator per component. Another way to think of this operator is as a "scope definition," because it brings elements of a path expression into scope so that these elements can be used in later operations or in predicate evaluation. The scoping rules are such that an object component gets into scope either by being scanned (captured using the logical Get operator in the leaves of expressions trees) or by being referenced (captured in the Mat operator). Components remain in scope until a projection discards them. The materialize operator allows a query processor to aggregate all component materializations required for the computation of a query, regardless of whether the components are needed for predicate evaluation or to produce the result of a query. The purpose of the materialize operator is to indicate to the optimizer where path expressions are used and where algebraic transformations can be applied. A number of transformation rules involving Mat are defined.

Path Indexes. Substantial research on object query optimization has been devoted to the design of index structures to speed up the computation of path expressions [Maier and Stein, 1986], [Bertino and Kim, 1989], [Valduriez, 1987], [Kemper and Moerkotte, 1994].

Computation of path expressions via indexes represents just one class of query-execution algorithms used in object-query optimization. In other words, efficient computation of path expressions through path indexes represents only one collection of implementation choices for algebraic operators, such as materialize and join, used to represent inter-object references. Section 14.6.3 describes a representative collection of query-execution algorithms that promise to provide a major benefit to the efficient execution of object queries. We will defer a discussion of some representative path index techniques to that section. Bertino and Kim [1989] present a more comprehensive survey of index techniques for object query optimization.

14.6.3 Query Execution

The relational DBMSs benefit from the close correspondence between the relational algebra operations and the access primitives of the storage system. Therefore, the generation of the execution plan for a query expression basically concerns the choice and implementation of the most efficient algorithms for executing individual algebra operators and their combinations. In object DBMSs, the issue is

more complicated due to the difference in the abstraction levels of behaviorally-defined objects and their storage. Encapsulation of objects, which hides their implementation details, and the storage of methods with objects pose a challenging design problem, which can be stated as follows: "At what point in query processing should the query optimizer access information regarding the storage of objects?" One alternative is to leave this to the object manager [Straube and Özsu, 1995]. Consequently, the query-execution plan is generated from the query expression is obtained at the end of the query-rewrite step by mapping the query expression to a well-defined set of object-manager interface calls. The object-manager interface consists of a set of execution algorithms. This section reviews some of the execution algorithms that are likely to be part of future high-performance object-query execution engines.

A query-execution engine requires three basic classes of algorithms on collections of objects: *collection scan*, *indexed scan*, and *collection matching*. Collection scan is a straightforward algorithm that sequentially accesses all objects in a collection. We will not discuss this algorithm further due to its simplicity. Indexed scan allows efficient access to selected objects in a collection through an index. It is possible to use an object's field or the values returned by some method as a key to an index. Also, it is possible to define indexes on values deeply nested in the structure of an object (i.e., path indexes). In this section we mention a representative sample of path-index proposals. Set-matching algorithms take multiple collections of objects as input and produce aggregate objects related by some criteria. Join, set intersection, and assembly are examples of algorithms in this category.

Path Indexes

As indicated earlier, support for path expressions is a feature that distinguishes object queries from relational ones. Many indexing techniques designed to accelerate the computation of path expressions have been proposed [Maier and Stein, 1986], [Bertino and Kim, 1989] based on the concept of join index [Valduriez, 1987].

One such path indexing technique, developed for the GemStone object DBMS, creates an index on each class traversed by a path [Maier and Stein, 1986]. This technique was also proposed for the Orion object DBMS [Bertino and Kim, 1989]. In addition to indexes on path expressions, it is possible to define indexes on objects across their type inheritance. Kim et al. [1989] provide a thorough discussion of such indexing techniques through inheritance.

Access support relations [Kemper and Moerkotte, 1994] are an alternative general technique to represent and compute path expressions. An access support relation is a data structure that stores selected path expressions. These path expressions are chosen to be the most frequently navigated ones. Studies provide initial evidence that the performance of queries executed using access support relations improves by about two orders of magnitude over queries that do not use access support relations. A system using access support relations must also consider the cost of maintaining them in the presence of updates to the underlying base relations.

Set Matching

As indicated earlier, path expressions are traversals along the composite object composition relationship. We have already seen that a possible way of executing a path expression is to transform it into a join between the source and target sets of objects. A number of different join algorithms have been proposed, such as hybrid-hash join or pointer-based hash join [Shekita and Carey, 1990]. The former uses the divide-and-conquer principle to recursively partition the two operand collections into buckets using a hash function on the join attribute. Each of these buckets may fit entirely in memory. Each pair of buckets is then joined in memory to produce the result. The pointer-based hash join is used when each object in one operand collection (call R) has a pointer to an object in the other operand collection (call S). The algorithm follows three steps, the first one being the partitioning of R in the same way as in the hybrid hash algorithm, except that it is partitioned by OID values rather than by join attribute. The set of objects S is not partitioned. In the second step, each partition R_i of R is joined with S by taking R_i and building a hash table for it in memory. The table is built by hashing each object $r \in R$ on the value of its pointer to its corresponding object in S. As a result, all R objects that reference the same page in S are grouped together in the same hash-table entry. Third, after the hash table for R_i is built, each of its entries is scanned. For each hash entry, the corresponding page in S is read, and all objects in R that reference that page are joined with the corresponding objects in S. These two algorithms are basically centralized algorithms, without any distributed counterparts. So we will not discuss them further.

An alternative method of join execution algorithm, *assembly* [Keller et al., 1991], is a generalization of the pointer-based hash-join algorithm for the case when we need to compute a multi-way join. Assembly has been proposed as an additional object algebra operator. This operation efficiently assembles the fragments of objects' states required for a particular processing step, and returns them as a complex object in memory. It translates the disk representations of complex objects into readily traversable memory representations.

Assembling a complex object rooted at objects of type R containing object components of types S, U, and T, is analogous to computing a four-way join of these sets. There is a difference between assembly and n-way pointer joins in that assembly does not need the entire collection of root objects to be scanned before producing a single result.

Instead of assembling a single complex object at a time, the assembly operator assembles a *window*, of size W, of complex objects simultaneously. As soon as any of these complex objects becomes assembled and passed up the query-execution tree, the assembly operator retrieves another one to work on. Using a window of complex objects increases the pool size of unresolved references and results in more options for optimization of disk accesses. Due to the randomness with which references are resolved, the assembly operator delivers assembled objects in random order up the query execution tree. This behavior is correct in set-oriented query processing, but may not be for other collection types, such as lists.

Example 14.9

Consider the example given in Figure 14.3, which assembles a set of Car objects. This is similar to the one given from Keller et al. [1991], but adapted to our example. The boxes in the figure represent instances of types indicated at the left, and the edges denote the composition relationships (e.g., there is an attribute of every object of type `Car` that points to an object of type `Engine`). Suppose that assembly is using a window of size 2. The assembly operator begins by filling the window with two (since $W = 2$) `Car` object references from the set (Figure 14.4a). The assembly operator begins

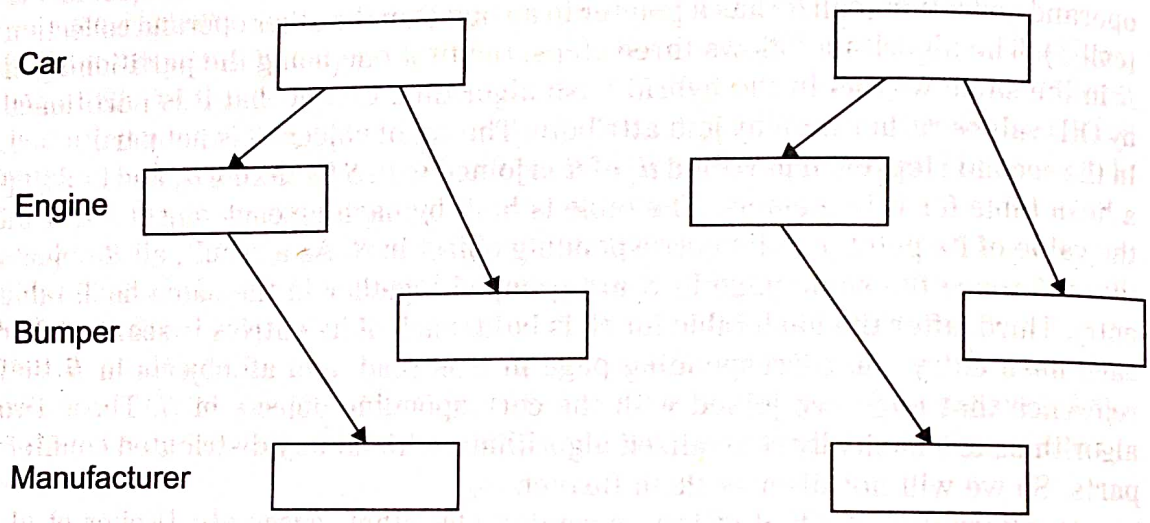

Figure 14.3. Two Assembled Complex Objects

by choosing among the current outstanding references, say $C1$. After resolving (fetching) $C1$, two new unresolved references are added to the list (Figure 14.4b). Resolving $C2$ results in two more references added to the list (Figure 14.4c), and so on until the first complex object is assembled (Figure 14.4g). At this point, the assembled object is passed up the query-execution tree, freeing some window space. A new Car object reference, $C3$, is added to the list and then resolved, bringing two new references $E3$, $B3$ (Figure 14.4h).

The objective of the assembly algorithm is to simultaneously assemble a window of complex objects. At each point in the algorithm, the outstanding reference that optimizes disk accesses is chosen. There are different orders, or schedules, in which references may be resolved, such as depth-first, breath-first, and elevator. Performance results indicate that elevator outperforms depth-first and breath-first under several data-clustering situations [Keller et al., 1991].

A number of possibilities exist in implementing a distributed version of this operation [Maier et al., 1994]. One strategy involves shipping all data to a central site for processing. This is straightforward to implement, but could be inefficient in general. A second strategy involves doing simple operations (e.g., selections, local assembly) at remote sites, then shipping all data to a central site for final

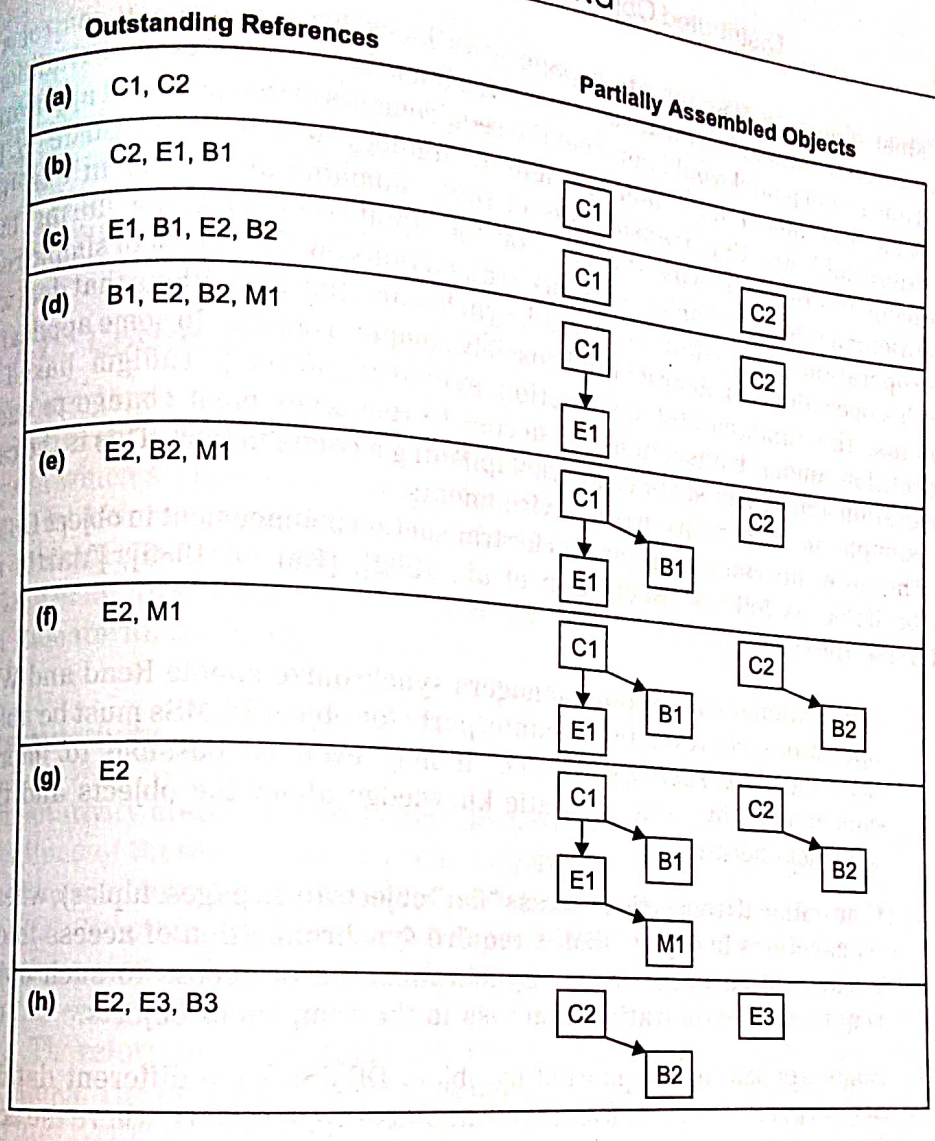

Figure 14.4. An Assembly Example

assembly. This strategy also requires fairly simple control, since all communication occurs through the central site. The third strategy is significantly more complicated: perform complex operations (e.g., joins, complete assembly of remote objects) at remote sites, then ship the results to the central site for final assembly. A distributed object DBMS may include all or some of these strategies.

14.7 TRANSACTION MANAGEMENT

Transaction management in *distributed* object DBMSs have not been studied except in relation to the cashing problem discussed earlier. However, transactions on objects raise a number of interesting issues, and their execution in a distributed environment can be quite challenging. This is an area which clearly requires more work. In this section we will discuss, at some length, the particular problems that arise in extending the transaction concept to object DBMSs.

Most object DBMSs maintain page level locks for concurrency control and support the traditional flat transaction model. It has been argued that the traditional flat transaction model would not meet the requirements of the advanced application domains that object data management technology would serve. Some of the considerations are that transactions in these domains are longer in duration, requiring interactions with the user or the application program during their execution. In the case of object systems, transactions do not consist of simple read/write operations, necessitating, instead, synchronization algorithms that deal with complex operations on abstract (and possibly complex) objects. In some application domains, the fundamental transaction synchronization paradigm based on competition among transactions for access to resources must change to one of cooperation among transactions in accomplishing a common task. This is the case, for example, in cooperative work environments.

The more important requirements for transaction management in object DBMSs can be listed as follows [Buchmann et al., 1992], [Kaiser, 1989], [Martin and Pedersen, 1994]:

1. Conventional transaction managers synchronize simple Read and Write operations. However, their counterparts for object DBMSs must be able to deal with *abstract operations*. It may even be possible to improve concurrency by using semantic knowledge about the objects and their abstract operations.

2. Conventional transactions access "flat" objects (e.g., pages, tuples), whereas transactions in object DBMSs require synchronization of access to composite and complex objects. Synchronization of access to such objects requires synchronization of access to the component objects.

3. Some applications supported by object DBMSs have different database access patterns than conventional database applications, where the access is competitive (e.g., two users accessing the same bank account). Instead, sharing is more cooperative, as in the case of, for example, multiple users accessing and working on the same design document. In this case, user accesses must be synchronized, but users are willing to cooperate rather than compete for access to shared objects.

4. These applications require the support of *long-running activities* spanning hours, days or even weeks (e.g., when working on a design object). Therefore, the transaction mechanism must support the sharing of partial results. Furthermore, to avoid the failure of a partial task jeopardizing a long activity, it is necessary to distinguish between those activities that are essential for the completion of a transaction and those that are not, and to provide for alternative actions in case the primary activity fails.

5. It has been argued that many of these applications would benefit from *active capabilities* for timely response to events and changes in the environment. This new database paradigm requires the monitoring of

events and the execution of system-triggered activities within running transactions.

These requirements point to a need to extend the traditional transaction management functions in order to capture application and data semantics, and to a need to relax isolation properties. This, in turn, requires revisiting every aspect of transaction management that we discussed in Chapters 10–12.

14.7.1 Correctness Criteria

In Chapter 11, we introduced serializability as the fundamental correctness criteria for concurrent execution of database transactions. There are a number of different ways in which serializability can be defined, even though we did not elaborate on this point before. These differences axe based on how a *conflict* is defined. We will concentrate on three alternatives: *commutativity* [Weihl, 1988], [Weihl, 1989], [Fekete et al., 1989], *invalidation* [Herlihy, 1990], and *recoverability* [Badrinath and Ramamritham, 1987].

Commutativity

Commutativity states that two operations conflict if the results of different serial executions of these operations are not equivalent. The traditional conflict definition discussed in Chapter 11 is a special case. Consider the simple operations $R(x)$ and $W(x)$. If nothing is known about the abstract semantics of the Read and Write operations or the object x upon which they operate, it has to be accepted that a $R(x)$ following a $W(x)$ does not retrieve the same value as it would prior to the $W(x)$. Therefore, a Write operation always conflicts with other Read or Write operations. The conflict table (or the compatibility matrix) given in Figure 11.5 for Read and Write operations is, in fact, derived from the commutativity relationship between these, two operations. This table was called the compatibility matrix in Chapter 11, since two operations that do not conflict are said to be compatible. Since this type of commutativity relies only on syntactic information about operations (i.e., that they are Read and Write), we call this *syntactic commutativity* [Buchmann et al., 1992].

In Figure 11.5, Read and Write operations and Write and Write operations do not commute. Therefore, they conflict, and serializability maintains that either all conflicting operations of transaction T_i precede all conflicting operations of T_k, or vice versa.

If the semantics of the operations are taken into account, however, it may be possible to provide a more relaxed definition of conflict. Specifically, some concurrent executions of Write-Write and Read-Write may be considered non-conflicting. *Semantic commutativity* (e.g., [Weihl, 1988], [Weihl, 1989]) makes use of the semantics of operations and their termination conditions.

Example 14.10

Consider, for example, an abstract data type set and three operations defined on it: Insert and Delete, which correspond to a Write, and Member, which tests for membership and corresponds to a Read. Due to the semantics of these operations, two Insert operations on an instance of set type would commute, allowing them to be executed concurrently. The commutativity of Insert with Member and the commutativity of Delete with Member depends upon whether or not they reference the same argument and their results[8].

It is also possible to define commutativity with reference to the database state. In this case, it is usually possible to permit more operations to commute.

Example 14.11

In Example 14.7, we indicated that an Insert and a Member would commute if they do not refer to the same argument. However, if the set already contains the referred element, these two operations would commute even if their arguments are the same.

The question now is how to formalize this intuitive understanding of commutativity of operations on abstract data types based on their semantics. We follow Weihl [1988,1989] in addressing this question.

The conflict relations defined in Weihl [1988] are binary relations between operations that consider both the operation and its result. An operation is now defined as a pair of invocation and response to that invocation; e.g., x: [Insert(3),ok] is a valid invocation of an insert operation on set x that returns that the operation was performed correctly. Two different kinds of commutativity and their corresponding commutativity relation can be defined: *forward commutativity* and *backward commutativity*. Assume two operations P and Q and a state s of an object. Forward commutativity is then defined as follows: For every state s in which P and Q are both defined (individually), $P(Q(s)) = Q(P(s))$ and $P(Q(s))$ is defined (i.e., it is not the null state). The notation used means that if we first apply operation

	[Insert(*i*),ok]	[Delete(*i*),ok]	[Member(*i*),true]	[Member(*i*),false]
[Insert(*i*),ok]	+	−	+	−
[Delete(*i*),ok]	−	+	+	+
[Member(*i*),true]	+	−	+	+
[Member(*i*),false]	−	+	+	+

Figure 14.5. Compatibility Table for Forward Commutativity in Sets

[8]Depending upon the operation, the result may either be a flag that indicates whether the operation was successful (for example, the result of Insert may be "OK") or the value that the operation returns (as in the case of a Read).

Q to states and then operation P to that result, we obtain the same result as if we apply first P to state s and then operation Q to the result. Backward commutativity is denned as follows: For every state s in which we know that $P(Q(s))$ is denned (i.e., $Q(s)$ is defined but $P(s)$ may or may not be defined), $P(Q(s)) = Q(P(a))$. Of course, both forward and backward commutativity extend to the case where P and Q are sequences of operations, rather than a single operation.

Example 14.12

The forward and backward compatibility relations for the set ADT are given in Figures 14.5 and 14.6, respectively. In these tables, the Member operation is defined once with a successful execution return code ("true"), and once with an unsuccessful execution code ("false").

It is important to notice the difference in the states over which the operations are defined. In forward commutativity, both operations are defined over the same initial state. Therefore, it makes no difference which operation is applied first, as long as the final result is the same.

	[Insert(*i*),ok]	[Delete(*i*),ok]	[Member(*i*),true]	[Member(*i*),false]
[Insert(*i*),ok]	+	–	–	–
[Delete(*i*),ok]	–	+	–	+
[Member(*i*),true]	–	–	+	+
[Member(*i*),false]	–	+	+	+

Figure 14.6. Compatibility Table for Backward Commutativity in Sets

Example 14.13

If the initial state of the set object is {1,2,3}, the first operation on that set object is the pair of invocation-response [Insert(3), ok], and the second operation is [Member(3), true], both operations are defined on {1,2,3}, and the result of applying them in either order is the same. However, if all we know is that the state is {1,2,3} after applying the operation [Insert(3), ok], we cannot say whether the initial state was {1,2} or {1,2,3}. Therefore, for a set object, the operations [Insert(x), ok] and [Member(x), true] do commute forward, but do not commute backwards.

Notice from Figures 14.5 and 14.6 that, for the set object as we defined it, the backward commutativity relation subsumes the forward commutativity relation. However, this is not true for all objects. In general, forward and backward commutativity relations are incomparable.

Example 14.14

This example demonstrates that the forward and backward commutativity relations are incomparable by considering these relations for a bank account abstract data type. The relations are given in Figures 14.7 and 14.8. The operations are self-explanatory, except for Post(i), which posts a given percentage i of interest to the account object. The argument of the operations is amounts of funds.

	[Withdraw(m),ok]	[Withdraw(m),no]	[Deposit(n),ok]	[Balance,r]	[Post(i),ok]
[Withdraw(m),ok]	–	+	+	–	–
[Withdraw(m),no]	+	+	–	+	+
[Deposit(n),ok]	+	–	+	–	–
[Balance,r]	–	+	–	+	–
[Post(i),ok]	–	+	–	–	–

Figure 14.7. Forward Commutativity Table for a Bank Acount Object

The incomparability of these two relations causes difficulties in implementing transaction managers that use them. Basically, one or the other must be chosen for enforcement, even though each permits certain operation histories which the other one rejects. Nakajima [1994] extends this work and defines a *general commutativity relation*, which is a superset of both the forward and the backward commutativity relations. If $FC(o)$ and $BC(o)$ are the forward and backward commutativity relations, respectively, for object o, then the general commutativity relation is defined as $GC(o) = FC(o) \cup BC(o)$. The general commutativity relation for the bank account example is given in Figure 14.9. Even though it seems preferable to use the general commutativity relation, enforcing it is not straightforward.

	[Withdraw(m),ok]	[Withdraw(m),no]	[Deposit(n),ok]	[Balance,r]	[Post(i),ok]
[Withdraw(m),ok]	+	–	–	–	–
[Withdraw(m),no]	–	+	–	+	+
[Deposit(n),ok]	–	–	+	–	–
[Balance,r]	–	+	–	+	–
[Post(i),ok]	–	+	–	–	–

Figure 14.8. Backward Commutativity Table for a Bank Acount Object

Invalidation

Invalidation [Herlihy, 1990] defines a conflict between two operations not on the basis of whether they commute or not, but according to whether or not the execution of one invalidates the other. An operation P invalidates another operation Q if there are two histories H_1 and H_2 such that $H_1 \bullet P \bullet H_2$ and $H_1 \bullet H_2 \bullet Q$ are legal, but $H_1 \bullet P \bullet H_2 \bullet Q$ is not. In this context, a *legal history* represents a correct history for the set object and is determined according to its semantics. Accordingly, an *invalidated-by* relation is defined as consisting of all operation pairs (P, Q) such that P invalidates Q. The invalidated-by relation establishes the conflict relation that forms the basis of establishing serializability. Considering the Set example, an Insert cannot be invalidated by any other operation, but a Member can be invalidated by a Delete if their arguments are the same.

	[Withdraw(*m*),ok]	[Withdraw(*m*),no]	[Deposit(*n*),ok]	[Balance,*r*]	[Post(*i*),ok]
[Withdraw(*m*),ok]	+	+	+	–	–
[Withdraw(*m*),no]	+	+	–	+	+
[Deposit(*n*),ok]	+	–	+	–	–
[Balance,*r*]	–	+	–	+	–
[Post(*i*),ok]	–	+	–	–	–

Figure 14.9. General Commutativity Relation for the Bank Account Example

Recoverability

Recoverability [Badrinath and Ramamritham, 1987] is another conflict relation that has been defined to determine serializable histories[9]. Intuitively, an operation P is said to be *recoverable with respect to* operation Q if the value returned by P is independent of whether Q executed before P or not. The conflict relation established on the basis of recoverability seems to be identical to that established by invalidation. However, this observation is based on only a few examples, and there is no formal proof of this equivalence. In fact, the absence of a formal theory to reason about these conflict relations is a serious deficiency that must be addressed.

14.7.2 Transaction Models and Object Structures

In Chapter 10, we considered a number of transaction models ranging from flat transactions to workflow systems. In our previous discussion, we did not consider the granularity of database objects upon which these transactions operate (we simply referred to it as a "lock unit"). Now we will consider the alternatives, and,

[9] Recoverability as used in [Badrinath and Ramamritham, 1987] is different from the notion of recoverability as we defined it in Chapter 12 and as found in [Bernstein et al., 1987] and [Hadzilacos, 1988].

together with the transaction model alternatives, they will provide us with a two-dimensional design space for possible transaction system implementations.

Along the object structure dimension, we identify *simple objects* (e.g., files, pages, records), objects as instances of *abstract data types* (ADTs), *full-fledged objects*, and *active objects* in increasing complexity.

Our previous discussion primarily considered systems that operate on simple objects, mostly physical pages. There are systems that provide for concurrency at the record level, but the overhead is usually high and record operations alone are not atomic, requiring synchronization at the page level as well. The characterizing feature of this class is that the operations on simple objects do not take into account the semantics of the objects. For example, an update of a page is considered a Write on the page, without considering what logical object is stored on the page.

From the perspective of transaction processing, ADTs introduce a need to deal with abstract operations, as we saw in the previous section. Abstract operations lend themselves nicely to the incorporation of their semantics into the definition of the correctness criterion. The execution of transactions on ADTs may require a multi-level mechanism as presented in Chapter 10 [Beeri et al., 1988], [Weikum, 1991]. In such systems, individual transactions represent the highest level of abstraction. The abstract operations constitute a lower level of abstraction, and are further decomposed into simple Reads and Writes at the lowest level. The correctness criterion, whatever it is, must be applied to each level individually.

We make the distinction between objects as instances of abstract data types and full-fledged objects to note that the latter have a complex structure (i.e., contain other objects), and that their types (classes) participate in a subtype (inheritance) lattice[10]. They must be treated separately due to a number of considerations:

1. Running a transaction against a composite object may actually spawn additional transactions on its component objects. This forces an *implicit nesting* [Badrinath and Ramamritham, 1988] on the transaction itself (as opposed to explicit nesting, which we discussed as part of transaction structure in Chapter 10). More importantly, the operations in these nestings are themselves abstract and need to be handled as multilevel transactions [Weikum and Hasse, 1993].

2. Subtyping/inheritance involves the sharing of behavior and/or state among objects. Therefore, the semantics of accessing an object at some level in the lattice must account for this.

We can also distinguish between *passive* and *active* objects. Although the approaches to the management of active objects vary, all proposals are similar in that active objects are capable of responding to events by triggering the execution of actions when certain conditions are satisfied. The events that are to be

[10]Strictly speaking, abstract data types can have complex structures. However, the transaction work on abstract data types has consistently assumed a "simple" ADT structure. Our reference to "objects as instances of ADTs" should be understood within this context and with this qualification.

monitored, the conditions that must be fulfilled, and the actions that are executed in response are typically defined in the form of event-condition-action (ECA) rules [Dayal et al., 1988], [Kotz et al., 1988]. Since events may be detected while executing a transaction on that object, the execution of the corresponding rule may be spawned as a nested transaction. Depending on the manner in which rules are coupled to the original transaction, different nestings may occur [Hsu et al., 1988]. The spawned transaction may execute immediately, it may be deferred to the end of the transaction, or it may execute in a separate transaction. Since additional rules may fire within a rule execution, nestings of arbitrary depth are possible. We will not consider active objects any further.

14.7.3 Transactions Management in Object DBMSs

As indicated above, transaction management in object DBMSs must deal with the composition (aggregation) graph, which shows the composite object structure, and *type (class) lattice*, which represents the **is-a** relationship between objects.

The aggregation graph requires methods for dealing with the synchronization of accesses to objects which have other objects as components. The class (type) lattice requires the transaction manager to take into account schema evolution concerns.

In addition to these structures, object DBMSs store methods together with data. Synchronization of shared access to objects must take into account method executions. In particular, transactions invoke methods which may, in turn, invoke other methods. Thus, even if the transaction model is flat, the execution of these transactions may be dynamically nested.

All of these factors introduce difficulties in executing transactions. Even the definition of conflicting operations becomes more involved. The conflict definition that we discussed earlier may no longer apply in object DBMSs. The classical conflict definition is based on the (non-)commutativity of operations that access the same object. In object DBMSs, there may be conflicts between operations that access different objects. This is due to the existence of the aggregation graph and the type lattice. Consider an operation O_1 which accesses object x, which has another object y as one of its components (i.e., x is a composite or complex object). There may be another operation O_2 (assume O_1 and O_2 belong to different transactions) which accesses y. According to the classical definition of a conflict, we would not consider O_1 and O_2 to be conflicting, since they access different objects. However, O_1 considers y as part of x and may want to access y while it accesses x, causing a conflict with O_2.

Schemes that are developed for object DBMSs must take these issues into consideration. In the remainder of this section, we will present some of the solutions that have been proposed.

Synchronizing Access to Objects

The inherent nesting in method invocations can be used to develop algorithms based on the well-known nested 2PL and nested timestamp ordering algorithms

[Hadzilakos and Hadzilakos, 1991]. In the process, intra-object parallelism may be exploited to improve concurrency. In other words, attributes of an object can be modeled as data elements in the database, whereas the methods are modeled as transactions enabling multiple invocations of an object's methods to be active simultaneously. This can provide more concurrency if special intra-object synchronization protocols can be devised which maintain the compatibility of synchronization decisions at each object.

Consequently, a method execution (modeled as a transaction) on an object consists of *local steps,* which correspond to the execution of local operations together with the results that are returned, and *method steps,* which are the method invocations together with the return values. A local operation is an atomic operation (such as Read, Write, Increment) that affects the object's variables. A method execution defines the partial order among these steps in the usual manner.

One of the fundamental directions of this work is to provide total freedom to objects in how they achieve intra-object synchronization. The only requirement is that they be "correct" executions, which, in this case, means that they should be serializable based on commutativity. As a result of the delegation of intra-object synchronization to individual objects, the concurrency control algorithm concentrates on inter-object synchronization.

An alternative approach based on multigranularity locking is used in Orion [Garza and Kim, 1988] and O_2[11] [Cart and Ferrie, 1990], even though they use different granularity hierarchies.

Multigranularity locking defines a hierarchy of lockable database granules (thus the name "granularity hierarchy") as depicted in Figure 14.10. In relational DBMSs, files correspond to relations and records correspond to tuples. In object DBMSs, the correspondence is with classes and instance objects, respectively.

The advantage of this hierarchy is that it addresses the tradeoff between coarse granularity locking and fine granularity locking. Coarse granularity locking (at the file level and above) has low locking overhead, since a small number of locks are set, but it significantly reduces concurrency. The reverse is true for fine granularity locking.

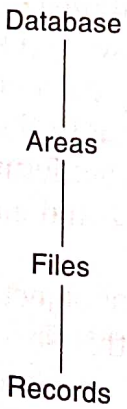

Figure 14.10. Multiple Granularities

[11] Even though this technique has been proposed for O_2, the commercial implementation of the system uses a straightforward page-level locking scheme.

The main idea behind multigranularity locking is that a transaction that locks at a coarse granularity implicitly locks all the corresponding objects of finer granularities. For example, explicit locking at the file level involves implicit locking of all the records in that file. To achieve this, two more lock types in addition to shared (S) and exclusive (X) are defined: *intention* (or *implicit*) *shared* (IS) and *intention* (or *implicit*) *exclusive* (IX). A transaction that wants to set an S or an IS lock on an object has to first set IS or IX locks on its ancestors (i.e., related objects of coarser, granularity). Similarly, a transaction that wants to set an X or an IX lock on an object must set IX locks on all of its ancestors. Intention locks cannot be released on an object if the descendants of that object are currently locked.

One additional complication arises when a transaction wants to read an object at some granularity and modify some of its objects at a finer granularity. In this case, both an S lock and an IX lock must be set on that object. For example, a transaction may read a file and update some records in that file (similarly, a transaction in object DBMSs may want to read the class definition and update some of the instance objects belonging to that class). To deal with these cases, a *shared intention exclusive* (SIX) lock is introduced, which is equivalent to holding an S and an IX lock on that object. The lock compatibility matrix for multigranularity locking is shown in Figure 14.11.

Orion's granularity hierarchy is shown in Figure 14.12. The lock modes that are supported and their compatibilities are exactly those given in Figure 14.11. Instance objects are locked only in S or X mode, while class objects can be locked in all five modes. The interpretation of these locks on class objects is as follows:

- S mode: Class definition is locked in S mode, and all its instances are implicitly locked in S mode. This prevents another transaction from updating the instances.

	S	X	IS	IX	SIX
S	+	–	+	–	–
X	–	–	–	–	–
IS	+	–	+	+	+
IX	–	–	+	+	–
SIX	–	–	+	–	–

Figure 14.11. Compatibility Table for Multigranularity Locking

- X mode: Class definition is locked in X mode, and all its instances are implicitly locked in X mode. Therefore, the class definition and all instances of the class may be read or updated.

- IS mode: Class definition is locked in IS mode, and the instances are to be locked in S mode as necessary.

- IX mode: Class definition is locked in IX mode, and the instances will be locked in either S or X mode as necessary.
- SIX mode: Class definition is locked in S mode, and all the instances are implicitly locked in S mode. Those instances that are to be updated are explicitly locked in X mode as the transaction updates them.

Management of Type Lattice

One of the important requirements of object DBMSs is dynamic schema evaluation. Consequently, systems must deal with transactions that access schema objects (i.e., types, classes, etc.), as well as instance objects. The existence of schema change operations intermixed with regular queries and transactions, as well as the (multiple) inheritance relationship defined among classes, complicates the picture. First, a query/transaction may not only access instances of a class, but may also access instances of subclasses of that class (i.e., *deep extent*). Second, in a composite object, the domain of an attribute is itself a class. So accessing an attribute of a class may involve accessing the objects in the sublattice rooted at the domain class of that attribute.

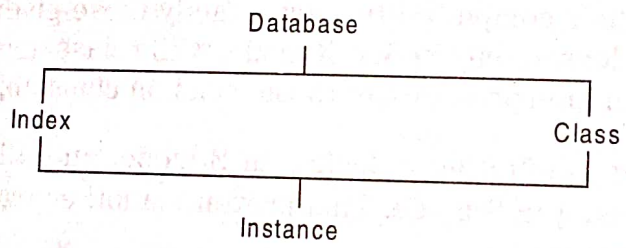

Figure 14.12. Orion's Granularity Hierarchy

One way to deal with these two problems is, again, by using multigranularity locking, as done in Orion. The straightforward extension of multigranularity locking where the accessed class and all its subclasses are locked in the appropriate mode does not work very well. This approach is inefficient when classes close to the root are accessed, since it involves too many locks. The problem may be overcome by introducing *read-lattice* (R) and *write-lattice* (W) lock modes, which not only lock the target class in S or X modes, respectively, but also implicitly lock all subclasses of that class in S and X modes, respectively. However, this solution does not work with multiple inheritance (which is the third problem).

The problem with multiple inheritance is that a class with multiple supertypes may be implicitly locked in incompatible modes by two transactions that place R and W locks on different superclasses. Since the locks on the common class are implicit, there is no way of recognizing that there is already a lock on the class. Thus, it is necessary to check the superclasses of a class that is being locked. Orion handles this by placing *explicit* locks, rather than implicit ones, on subclasses.

Consider the type lattice of Figure 14.13, which is simplified from Garza and Kim [1988]. If transaction T_1 sets an IR lock on class A and an R lock on C, it also sets an explicit R lock on E. When another transaction T_2 places an IW lock on F and a W lock on G, it will attempt to place an explicit W lock on E. However, since there is already an R lock on E, this request will be rejected.

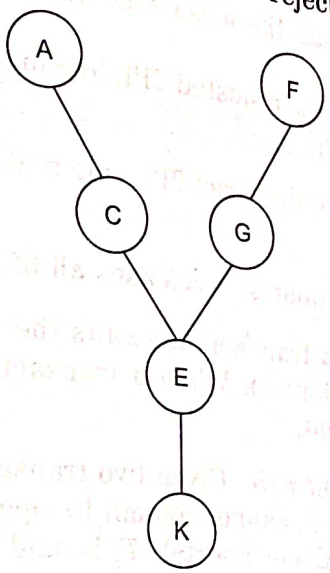

Figure 14.13. An Example Class Lattice

The approach followed by Orion sets explicit locks on subclasses of a class that is being modified. An alternative, which sets locks at a finer granularity, uses ordered sharing, as discussed in Chapter 11 [Agrawal and El-Abbadi, 1994]. In a sense, the algorithm is an extension of Weird's commutativity-based approach to object DBMSs using a nested transaction model.

Classes are modeled as objects in the system similar to reflective systems that represent schema objects as first-class objects. Consequently, methods can be defined that operate on class objects: $add(m)$ to add method m to the class, $del(m)$ to delete method m from the class, $rep(m)$ to replace the implementation of method m with another one, and $use(m)$ to execute method m. Similarly, atomic operations are defined for accessing attributes of a class. These are identical to the method operations with the appropriate change in semantics to reflect attribute access. The interesting point to note here is that the definition of the *use* (a) operation for attribute a indicates that the access of a transaction to attribute a within a method execution is through the *use* operation. This requires that each method explicitly lists all the attributes that it accesses. Thus, the following is the sequence of steps that are followed by a transaction, T, in executing a method m:

1. Transaction T issues operation $use(m)$.
2. For each attribute a that is accessed by method m, T issues operation $use(a)$.
3. Transaction T invokes method m.

Commutativity tables are defined for the method and attribute operations. Based on the commutativity tables, ordered sharing lock tables for each atomic operation are determined (see Chapter 11). Specifically, a lock for an atomic operation p has a shared relationship with all the locks associated with operations with which p has a non-conflicting relationship, whereas it has an ordered shared relationship with respect to all the locks associated with operations with which p has a conflicting relation.

Based on these lock tables, a nested 2PL locking algorithm is used with the following considerations:

1. Transactions observe the strict 2PL rule and hold on to their locks until termination.
2. When a transaction aborts, it releases all of its locks.
3. The termination of a transaction awaits the termination of its children (closed nesting semantics). When a transaction commits, its locks are inherited by its parent.
4. *Ordered commitment rule.* Given two transactions T_i and T_j such that T_i is *waiting for* T_j, T_i cannot commit its operations on any object until T_j terminates (commits or aborts). T_i is said to be *waiting-for* T_j if:
   - T_i is not the root of the nested transaction and T_i was granted a lock in ordered shared relationship with respect to a lock held by T_j on an object such that T_j is a descendent of the parent of T_i; or
   - T_i is the root of the nested transaction and T_i holds a lock (that it has inherited or it was granted) on an object in ordered shared relationship with respect to a lock held by T_3 or its descendants.

Management of Composition (Aggregation) Graph

Studies dealing with the composition graph are more prevalent. The requirement for object DBMSs to model composite objects in an efficient manner has resulted in considerable interest in this problem.

We start the discussion in this section with an overview of Orion's approach to managing the aggregation hierarchy, which is, once more, based on multigranularity locking. Two alternatives are identified. One is, as suggested in the previous section, to lock a composite object and all the classes of the component objects. This is clearly unacceptable, since it involves locking the entire composite object hierarchy, thereby restricting performance significantly. The second alternative is to lock the component object instances within a composite object. In this case, it is necessary to chase all the references and lock all those objects. This is quite cumbersome, since it involves locking so many objects.

The problem is that the multigranularity locking protocol does not recognize the composite object as one lockable unit. To overcome this problem, three new

lock modes are introduced: ISO, IXO, and SIXO, corresponding to the IS, IX, and SIX modes, respectively. These lock modes are used for locking component classes of a composite object. The compatibility of these modes is shown in Figure 14.14. The protocol is then as follows: to lock a composite object, the root class is locked in X, IS, IX, or SIX mode, and each of the component classes of the composite object hierarchy is locked in the X, ISO, IXO, and SIXO mode, respectively.

	S	X	IS	IX	SIX	ISO	IXO	SIXO
S	+	–	+	–	–	+	–	–
X	–	–	–	–	–	–	–	–
IS	+	–	+	+	+	+	–	–
IX	–	–	+	+	–	–	–	–
SIX	–	–	+	–	–	–	–	–
ISO	+	–	+	+	–	+	+	+
IXO	–	–	–	–	–	+	+	–
SIXO	N	N	N	N	N	Y	N	N

Figure 14.14. Compatibility Matrix for Composite Objects

The Orion concurrency control algorithms, based on multigranularity locking, enforce serializability. The recovery algorithm used is logging using no-fix/flush, which requires undo but not redo.

An extension of multigranularity locking to deal with aggregation graph is discussed in Herrmann et al. [1990]. The extension has to do with the replacement of a single static lock graph with a hierarchy of graphs associated with each type and query. There is a "general lock graph" which controls the entire process (Figure 14.15). The smallest lockable units are called *basic lockable units* (BLU). A number of BLUs can make up a *homogeneous lockable unit* (HoLU), which consists of data of the same type. Similarly, they can make up a *heterogeneous lockable unit* (HeLU), which is composed of objects of different types. HeLUs can contain other HeLUs or HoLUs, indicating that component objects do not all have to be atomic. Similarly, HoLUs can consist of other HoLUs or HeLUs, as long as they are of the same type. The separation between HoLUs and HeLUs is meant to optimize lock requests. For example, a set of lists of integers is, from the viewpoint of lock managers, treated as a HoLU composed of HoLUs, which, in turn, consist of BLUs. As a result, it is possible to lock the whole set, exactly one of the lists, or even just one integer.

At type definition time, an object-specific lock graph is created which obeys the general lock graph. As a third component, a query-specific lock graph is generated during query (transaction) analysis. During the execution of the query (transaction), the query-specific lock graph is used to request locks from the lock

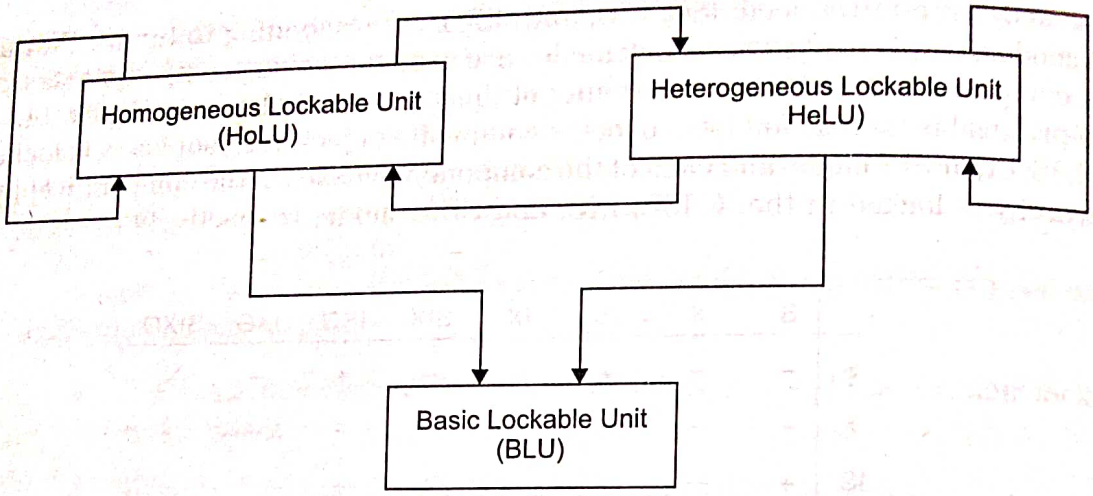

Figure 14.15. General Lock Graph

manager, which uses the object-specific lock graph to make the decision. The lock modes used are the standard ones (i.e., IS, IX, S, X).

Badrinath and Ramamritham [1988] discuss an alternative to dealing with composite object hierarchy based on commutativity. A number of different operations are defined on the aggregation graph:

1. Examine the contents of a vertex (which is a class).
2. Examine an edge (composed-of relationship).
3. Insert a vertex and the associated edge.
4. Delete a vertex and the associated edge.
5. Insert an edge.

Note that some of these operations (1 and 2) correspond to existing object operators, while others (3—5) represent schema operations.

Based on these operations, an *affected-set* can be defined for granularity graphs to form the basis for determining which operations can execute concurrently. The affected-set of a granularity graph consists of the union of:

- *edge-set*, which is the set of pairs (e, a) where e is an edge and a is an operation affecting e and can be one of *insert*, *delete*, *examine*
- *vertex-set*, which is the set of pairs (v, a), where v is a vertex and a is an operation affecting v and can be one of *insert*, *delete*, *examine*, or *modify*.

Using the affected-set generated by two transactions T_i and T_j of an aggregation graph, one may define whether T_i and T_j can execute concurrently or not. Commutativity is used as the basis of the conflict relation. Thus, two transactions T_i and T_j commute on object K if $\textit{affected-set}(T_i) \cap_K \textit{affected-set}(T_j) = \phi$.

These protocols synchronize on the basis of objects, not operations on objects. It may be possible to improve concurrency by developing techniques that synchronize operation invocations rather than locking entire objects.

Another semantics-based approach is described in Muth et al. [1993]. The distinguishing characteristics of this approach are the following:

1. Access to component objects are permitted without going through a hierarchy of objects (i.e., no multigranularity locking).
2. The semantics of operations are taken into consideration by a priori specification of method commutativities[12].
3. Methods invoked by a transaction can themselves invoke other methods. This results in a (dynamic) nested transaction execution, even if the transaction is syntactically flat;

The transaction model used to support (3) is open nesting, specifically multilevel transactions as described in Chapter 10. The restrictions imposed on the dynamic transaction nesting are:

- All pairs (p, g) of potentially conflicting operations on the same object have the same depth in their invocation trees
- For each pair (f', g') of ancestors of f and g whose depth of invocation trees are the same, f' and g' operate on the same object.

With these restrictions, the algorithm is quite straightforward. A semantic lock is associated with each method, and a commutativity table defines whether or not the various semantic locks are compatible. Transactions acquire these semantic locks before the invocation of methods, and they are released at the end of the execution of a subtransaction (method), exposing their results to others. However, the parents of committed subtransactions have a higher-level semantic lock, which restricts the results of committed subtransactions only to those that commute with the root of the subtransaction. This requires the definition of a semantic conflict test, which operates on the invocation hierarchies using the commutativity tables.

An important complication arises with respect to the two conditions outlined above. It is not reasonable to restrict the applicability of the protocol to only those for which those conditions hold. What has been proposed to resolve the difficulty is to give up some of the openness and convert the locks that were to be released at the end of a subtransaction into *retained locks* held by the parent. A number of conditions under which retained locks can be discarded for additional concurrency.

A very similar, but more restrictive, approach is discussed in Weikum and Hasse [1993]. The multilevel transaction model is used, but restricted to only two levels: the object level and the underlying page level. Therefore, the dynamic nesting that occurs when transactions invoke methods which invoke other methods

[12]The commutativity test employed in this study is state-independent. It takes into account the actual parameters of operations, but not the states. This is in contrast to Weihl's work [Weihl, 1988].

is not considered. The similarity with the above work is that page level locks are released at the end of the subtransaction, whereas the object level locks (which are semantically richer) are retained until the transaction terminates.

In both of the above approaches [Muth et al., 1993], [Weikum and Hasse, 1993], recovery cannot be performed by page-level state-oriented protocols. Since subtransactions release their locks and make their results visible, compensating transactions must be run to "undo" actions of committed subtransactions.

14.7.4 Transactions as Objects

One important characteristic of relational data model is its lack of a clear update semantics. The model, as it was originally defined, clearly spells out how the data in a relational database is to be retrieved (by means of the relational algebra operators), but does not specify what it really means to update the database. The consequence is that the consistency definitions and the transaction management techniques are orthogonal to the data model. It is possible—and indeed it is common—to apply the same techniques to non-relational DBMSs, or even to non-DBMS storage systems.

The independence of the developed techniques from the data model may be considered an advantage, since the effort can be amortized over a number of different applications. Indeed, the existing transaction management work on object DBMSs have exploited this independence by porting the well-known techniques over to the new system structures. During this porting process, the peculiarities of object DBMSs, such as class (type) lattice structures, composite objects and object groupings (class extents) are considered, but the techniques are essentially the same.

It may be argued that in object DBMSs, it is not only desirable but indeed essential to model update semantics within the object model.

REVIEW QUESTIONS

14.1 Explain fundamental object concepts and models.
14.2 Give an example for abstract data types.
14.3 What do you mean by horizontal class partitioning?
14.4 What do you mean by vertical class partitioning?
14.5 What are class-partitioning algorithms?
14.6 What are architectural issues in distributed object DBMS?
14.7 How do you perform object management?
14.8 What do you mean by distributed object storage?
14.9 Take an assembly example in object query processing and explain.
14.10 Explain transaction management through examples.