Chapter 15
Distributed Object Database Management

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. Relational databases have proven to be very successful in supporting business data processing applications. However, there are many applications for which relational systems may not be appropriate. Examples include XML data management, computer-aided design (CAD), office information systems (OIS), document management systems, and multimedia information systems. For these applications, different data models and languages are more suitable. Object database management systems (object DBMSs) are better candidates for the development of some of these applications due to the following characteristics [Özsu et al., 1994b]:

1. These 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 application 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 lan-

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_15, © Springer Science+Business Media, LLC 2011

guage, 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 type systems of 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 is typically record-at-a-time. This has resulted in the development of DBMS functions, such as cursor processing, that enable iterating over the sets of data objects retrieved by query languages. In an object system, complex database applications may be written entirely in a single object database programming language.

The main issue in object DBMSs is to improve application programmer productivity by overcoming the impedence mismatch problem with acceptable performance. It can be argued that the above requirements can be met by relational DBMSs, since one can possibly map them to relational data structures. In a strict sense this is true; however, from a modeling perspective, it makes little sense, since it forces programmers to map semantically richer and structurally complex objects that they deal with in the application domain to simple structures in representation.

Another alternative is to extend relational DBMSs with "object-oriented" functionality. This has been done, leading to "object-relational DBMS" [Stonebraker and Brown, 1999; Date and Darwen, 1998]. Many (not all) of the problems in object-relational DBMSs are similar to their counterparts in object DBMSs. Therefore, in this chapter we focus on the issues that need to be addressed in object DBMSs.

A careful study of the advanced applications mentioned above indicates that they are inherently distributed, and require distributed data management support. This gives rise to distributed object DBMSs, which is the subject of this chapter.

In Section 15.1, we provide the necessary background of the fundamental object concepts and issues in developing object models. In Section 15.2, we consider the distribution design of object databases. Section 15.3 is devoted to the discussion of the various distributed object DBMS architectural issues. In Section 15.4, we present the new issues that arise in the management of objects, and in Section 15.5 the focus is on object storage considerations. Sections 15.6 and 15.7 are devoted to fundamental DBMS functions: query processing and transaction management. These issues take interesting twists when considered within the context of this new technology; unfortunately, most of the existing work in these areas concentrate on non-distributed object DBMSs. We, therefore, provide a brief overview and some discussion of distribution issues.

We note that the focus in this chapter is on fundamental object DBMS technology. We do not discuss related issues such as Java Data Objects (JDO), the use of object models in XML work (in particular the DOM object interface), or Service Oriented Architectures (SOA) that use object technology. These require more elaborate treatment than we have room in this chapter.

15.1 Fundamental Object Concepts and Object Models

An object DBMS is a system that uses an "object" as the fundamental modeling and access primitive. There has been considerable discussion on the elements of an object DBMS [Atkinson et al., 1989; Stonebraker et al., 1990] as well as significant amount of work on defining an "object model". Although some have questioned whether it is feasible to define an object model, in the same sense as the relational model [Maier, 1989], a number of object models have been proposed. 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. Some standard object model specifications have emerged as part of language standards, the most important of which is that developed by the Object Data Management Group (ODMG) that includes an object model (commonly referred to as the ODMG model), an Object Definition Language (ODL), and an Object Query Language (OQL)[1] [Cattell et al., 2000]. As an alternative, there has been a proposal for extending the relational model in SQL3 (now known as SQL:1999) [Melton, 2002]. There has also been a substantial amount of work on the foundations of object models [Abadi and Cardelli, 1996; Abiteboul and Beeri, 1995; Abiteboul and Kanellakis, 1998a]. In the remainder of this section, we will review some of the design issues and alternatives in defining an object model.

15.1.1 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 tiple ⟨OID, state, interface⟩, in which OID is the object identifier, the corresponding state is some representation of the current state of the object, and the interface defines the behavior of the object. Let us consider these in turn.

Object identifier is an invariant property of an object which permanently distinguishes it logically and physically from all other objects, regardless of its state [Khoshafian and Copeland, 1986]. This enables referential object sharing [Khoshafian and Valduriez, 1987], which is the basis for supporting composite and complex (i.e., graph) structures (see Section 15.1.3). 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.

The *state* of an object is commonly defined as either an atomic value or a constructed value (e.g., tuple or set). Let D be the union of the system-defined domains

[1] The ODMG was an industrial consortium that completed its work on object data management standards in 2001 and disbanded. There are a number of systems now that conform to the developed standard listed here: http://www.barryandassociates.com/odmg-compliance.html.

(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 15.1. 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_1$0])

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 identifier (i.e., the identity of the object), which is immutable. This is analogous to updates in imperative programming languages in which object identifier is implemented by main memory pointers. However, object identifier is more general than pointers in the sense that it persists following the program termination. Another implication of object identifier is that objects may be shared without incurring the problem of data redundancy. We will discuss this further in Section 15.1.3.

Example 15.2. 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 i_1 (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*) that 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.

The *interface* of an object consist of its properties. These properties include instance variables that reflect the state of the object, and the methods that define the operations that can be performed on this object. All instance variables and all methods of an object do not need to be visible to the “outside world”. An object’s *public interface* may consist of a subset of its instance variables and methods.

Some object models take a uniform and behavioral approach. In these models, the distinction between values and objects are eliminated and everything is an object, providing uniformity, and there is no differentiation between intance variables and methods – there are only methods (usually called behaviors) [Dayal, 1989; Özsu et al., 1995a].

An important distinction emerges from the foregoing discussion between relational model and object models. Relational databases deal with data values in a uniform fashion. Attribute values are the atoms with which structured values (tuples and relations) may be constructed. In a value-based data model, such as the relational model, data are identified by values. A relation is identified by a name, and a tuple is identified by a key, a combination of values. In object models, by contrast, data are identified by its OID. This distinction is crucial; modeling of relationships among data leads to data redundancy or the introduction of foreign keys in the relational model. The automatic management of foreign keys requires the support of integrity constraints (referential integrity).

Example 15.3. Consider Example 15.2. 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. ♦

15.1.2 Types and Classes

The terms "type" and "class" have caused confusion as they have sometimes been used interchangeably and sometimes to mean different things. In this chapter, we will use the more common term "class" when we refer to the specific object model construct and the term "type" to refer to a domain of objects (e.g., integer, string).

A class is a template for a group of objects, thus defining a common type for these objects that conform to the template. 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. A class describes the type of data by providing a domain of data with the same structure, as well as methods applicable to elements of that domain. The abstraction capability of classes, commonly referred to as *encapsulation*, hides the implementation details of the methods, which can be written in a general-purpose programming language. As indicated earlier, some (possibly proper) subset of its class structure and methods make up the publicly visible interface of objects that belong to that class.

Example 15.4. 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. In our examples, we will use an abstract syntax. ODMG ODL is considerably more powerful than the syntax we use, but it is also more complicated, which is not necessary to demonstrate the concepts. The type definition of `Car` can be as follows using this abstract 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
     replaceTire(place, tire)
```

The class definition specifies that `Car` has eight attributes and two 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.

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. By contrast, replaceTire method requires users to provide two external arguments: place (where the tire replacement was done), and tire (which tire was replaced). ♦

The interface data structure of a class may be arbitrarily complex or large. For example, Car class 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 a class with operations specific to document manipulation.

A class has an *extent* that is the collection of all objects that conform to the class specification. In some cases, a class extent can be materialized and maintained, but this is not a requirement for all classes.

Classes 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, class operations capture parts of the application programs that are more closely associated with data. Therefore, it becomes possible to model 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.

We end this section with the introduction of another concept, collection, that appears explicitly in some object models. A *collection* is a grouping of objects. In this sense, a class extent is a particular type of collection – one that gathers all objects that conform to a class. However, collections may be more general and may be based on user-defined predicates. The results of queries, for example, are collections of objects. Most object models do not have an explicit collection concept, but it can be argued that they are useful [Beeri, 1990], in particular since collections provide for a clear closure semantics of the query models and facilitate definition of user views. We will return to the relationship between classes and collections after we introduce subtyping and inheritance 15.1.4.

15.1.3 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 15.3, 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 15.5. Let us revise Example 15.3 as follows. Assume that c_1 is one instance of Car type that is defined in Example 15.3. 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.

15.1.4 Subclassing and Inheritance

Object systems provide extensibility by allowing user-defined classes to be defined and managed by the system. This is accomplished in two ways: by the definition of classes using type constructors or by the definition of classes based on existing classes through the process of *subclassing*[3]. Subclassing is based on the *specialization* relationship among classes (or types that they define). A class A is a *specialization* of another class B if its interface is a superset of B's interface. Thus, a specialized class is more defined (or more specified) than the class from which it is specialized. A class may be a specialization of a number of classes; it is explicitly specified as a *subclass* of a subset of them. Some object models require that a class is specified as a subclass of only one class, in which case the model supports *single subclassing*; others allow *multiple subclassing*, where a class may be specified as a subclass of more than one class. Subclassing and specialization indicate an **is-a** relationship between classes (types). In the above example, A **is-a** B, resulting in *substitutability*: an instance of a subclass (A) can be substituted in place of an instance of any of its *superclasses* (B) in any expression.

[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.

[3] This is also referred to as *subtyping*. We use the term "subclassing" to be consistent with our use of terminology. However, recall from Section 15.1.2 that each class defines a type; hence the term "subtyping" is also appropriate.

If multiple subclassing is supported, the class system forms a semilattice that can be represented as a graph. In many cases, there is a single root of the class system, which is the least specified class. However, multiple roots are possible, as in C++ [Stroustrup, 1986], resulting in a class system with multiple graphs. If only single subclasssing is allowed, as in Smalltalk [Goldberg and Robson, 1983], the class system is a tree. Some systems also define a most specified type, which forms the bottom of a full lattice. In these graphs/trees, there is an edge from type (class) `A` to type (class) `B` if `A` is a subtype of `B`.

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

Declaring a class to be a subclass of another results in *inheritance*. If class `A` is a subclass of `B`, then its its properties consist of the properties that it natively defines as well as the properties that it inherits from `B`. Inheritance allows reuse. A subclass may inherit either the behavior (interface) of its superclass, or its implementation, or both. We talk of single inheritance and multiple inheritance based on the subclass relationship between the types.

Example 15.6. 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 subclass of `Object` that we assume is the root of the class lattice with common methods such as `Put` or `Store`. Vehicle is defined with five attributes and one method that 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 15.3. `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 15.3. This is because `Car` **is-a** `Vehicle`, and therefore inherits the attributes and methods of `Vehicle`. ♦

Subclassing and inheritance allows us to discuss an issue related to classes and collections. As we defined in Section 15.1.2, each class extent is a collection of objects that conform to that class definition. With subclassing, we need to be careful – the class extent consists of the objects that immediately conform to its definition, which is referred to as (*shallow extent*), along with the extensions of its subtypes (*deep extent*). For example in Example 15.6, the extent of `Vehicle` class consists of all vehicle objects (shallow extent) as well as all car objects (deep extent of `Vehicle`). One consequence of this is that the objects in the extent of a class are homogeneous with respect to subclassing and inheritance – they are all of the superclass's type. In contrast, a user-defined collection may be heterogeneous in that it can contain objects of types unrelated by subclassing.

15.2 Object Distribution Design

Recall from Chapter 3 that the two important aspects of distribution design are fragmentation and allocation. In this section we consider the analogue, in object databases, of the distribution design problem.

Distribution design in the object world brings new complexities due to the encapsulation of methods together with object state. An object is defined by its state and its methods. We can fragment the state, the method definitions, and the method implementation. Furthermore, the objects in a class extent can also be fragmented and placed at different sites. Each of these raise interesting problems and issues. For example, if fragmentation is performed only on state, are the methods duplicated with each fragment, or can one fragment methods as well? The location of objects with respect to their class definition becomes an issue, as does the type of attributes (instance variables). As discussed in Section 15.1.3, 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 method definitions are fragmented 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 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 15.2.1. 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 15.2.3. In the remainder, for simplicity, we assume a class-based object model that does not distinguish between types and classes.

15.2.1 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. Fragmenation 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 that 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 15.7. Consider the definition of the `Engine` class that is referred to in Example 15.6:

```
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: horsepower ≤ 150
p_2: 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 (i.e,. a class that cannot have any objects in its shallow extent). The objects of `Engine` class are distributed to the `Engine1` and `Engine2` classes based on the value of their horsepower attribute value. ♦

We should first note that this example points to a significant advantage of object models – we can explicitly state that methods in `Engine1` class mention only those with horsepower less-than-or-equal-to 150. Consequently, we are able to make distribution explicit (with state and behavior) that is not possible in the relational model.

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 propagation 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 quantitative information about the distributed object database. Therefore, a more precise approach would be 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 3: 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 3.

So far, we have considered fragmentation with respect to attributes only, because the methods were simple. Let us now consider complex methods; these 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.

15.2.2 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, 1995, 1998]. However, the break-up of encapsulation during vertical fragmentation has created significant doubts as to the suitability of vertical fragmentation in object DBMSs.

15.2.3 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.

15.2.4 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 that 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.

15.2.4.1 Affinity-based Approach

As covered in Section 3.3.2, 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. Horizontal and vertical class partitioning algorithms have been developed that are based on classifying instance variables and methods as being either simple or complex [Ezeife and Barker, 1995]. A complex instance variable is an object-based instance variable and is part of the class composition hierarchy. An alternative is a method-induced partitioning scheme, which applies the method semantics and appropriately generates fragments that match the methods data requirements [Karlapalem et al., 1996a].

15.2.4.2 Cost-Driven Approach

Though the affinity-based approach provides "intuitively" appealing partitioning schemes, it has been shown that these partitioning schemes do not always result in the greatest reduction of disk accesses required to process a set of applications [Florescu et al., 1997]. Therefore, a cost model for the number of disk accesses for processing both queries [Florescu et al., 1997] and methods [Fung et al., 1996] on an object oriented database has been developed. Further, an heuristic "hill-climbing" approach that 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.

15.2.5 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 that need to access multiple classes at different sites is a problem that 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 interdependency between methods and classes. The issue has been addressed by means of an iterative solution for methods and class allocation [Bellatreche et al., 1998].

15.2.6 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.

15.3 Architectural Issues

The preferred architectural model for object DBMSs has been client/server. We had discussed the advantages of these systems in Chapter 1. The design issues related to these systems are somewhat more complicated due to the characteristics of object models. The major 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 are 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 15.4; and storage management issues (object clustering and garbage collection) are considered in Section 15.5.

15.3.1 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 are shipped between the clients and the servers, and partly on the functionality provided to the clients and servers.

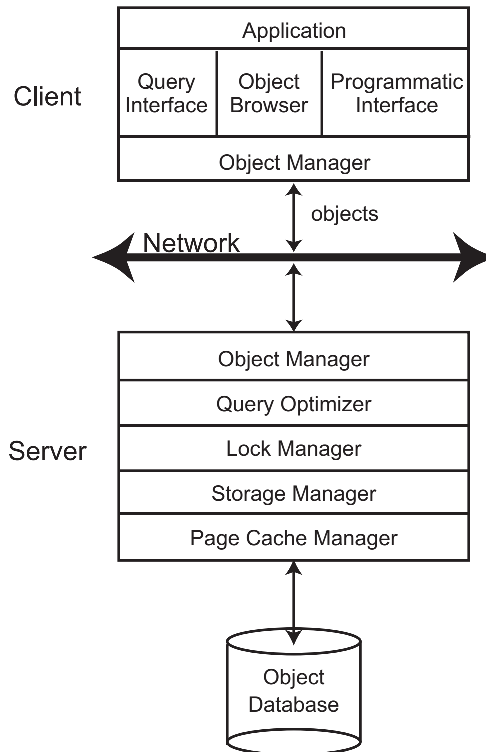

Fig. 15.1 Object Server Architecture

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 15.1). 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 15.4). 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 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 then the groups correspond to contiguous space on a disk page [Gerlhof and Kemper, 1994]. 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 15.2). 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 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.

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

[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.

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 15.4.2), since the unit of operation is uniformly a page.

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. Note that the concern here is not the relative cost of sending one object versus one page, but that of filtering objects at the server and sending them versus sending all of the pages on which these objects may reside. This is indeed what the relational client/server systems 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 (RPC) 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 is what is done in Web access, where the server simply serves as storage. Code shipping may be cheaper than data shipping. 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.

Clearly, both of these architectures have important advantages and limitations. There are systems that can shift from one architecture to the other – for example, O_2 would operate as a page server, but if the conflicts on pages increase, would shift to object shipping. Unfortunately, the existing performance studies do not establish clear tradeoffs, even though they provide interesting insights. The issue is complicated further by the fact that some objects, such as multimedia documents, may span multiple pages.

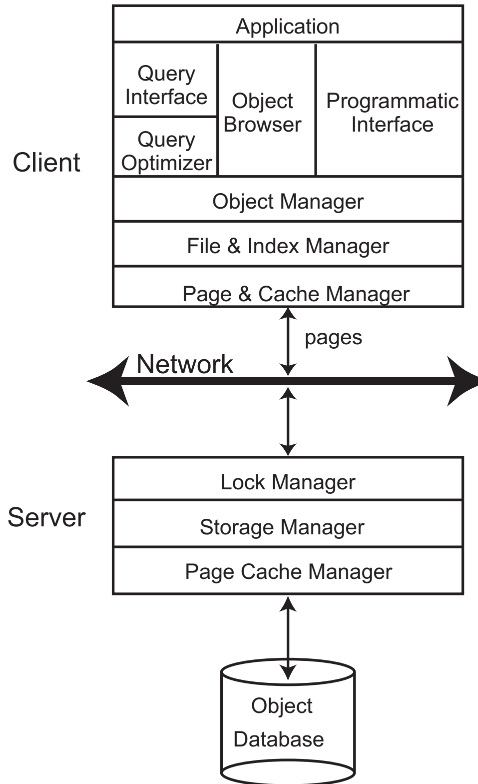

Fig. 15.2 Page Server Architecture

15.3.1.1 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 Kossmann, 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 site 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].

15.3.1.2 Server Buffer Management

The server buffer management issues in object client/server systems are not much different than their relational counterparts, 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., 1992]. 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.

15.3.2 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 (see Chapter 11), 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 to 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 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., 1997] because it increases the level of concurrency.

[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.

We 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, which 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 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) [Adya et al., 1995]. 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.

15.4 Object Management

Object management 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.

15.4.1 Object Identifier Management

As indicated in Section 15.1, 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 identifier 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 system-wide unique OID (i.e., a surrogate) per object. LOIDs can be generated either by using a system-wide unique counter (called pure LOID) or by concatenating a server identifier with a counter at each server (called pseudo-LOID). 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. Object-oriented database systems tend to prefer the logical identifier approach, which better supports dynamic environments.

Implementing transient object identifier involves the techniques used in programming languages. As for persistent object identifier, 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. The logical 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. For example, supporting object-sharing explicitly requires the implementation of object identifiers for all objects within the object manager and maintaining the sharing relationship. However, object identifiers for small objects can make the OID table quite large. If object sharing is not supported at the object manager level, but left to the higher levels of system (e.g., the compiler of the database language), more efficiency may be gained. Object identifier management is closely related to object storage techniques, which we will discuss in Section 15.5.

In distributed object DBMSs, it is 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 and is unique within a particular server. Sequence numbers are usually not re-used to prevent anomalies: an object o_i is deleted, and its sequence number is subsequently assigned to a newly created object o_j, but existing references to o_i now point to the new object o_j, which is not intended.
- **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 that 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 [Eickler et al.,

1995]. Hash tables provide fast access, but are not scalable as the database size increases. B^+-trees are scalable, but have logarithmic access time, and require complex concurrency control and recovery strategies. B^+-trees also support range queries, facilitating easy access to a collection of objects.

15.4.2 Pointer Swizzling

In object systems, one can navigate from one object to another using *path expressions* that involve attributes with object-based values. For example, if object `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, 1992]. 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 so that a pointer is swizzled to point to a location in the object table – that is LOIDs are used. 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 hardware-based scheme is that it leads to 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.

---

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

15.4.3 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. It is possible to move the object's state without moving its methods. 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 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* that we will consider under query processing in Section 15.6.3.

15.5 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 due to reference-based sharing. Indeed, in many object DBMSs, the only way to delete an object is to delete all references to it. Thus, object deletion and subsequent storage reclamation are critical and require special care.

15.5.0.1 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 15.1, 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 identifier implementation (i.e, LOID vs. POID). LOIDs incur more overhead (an indirection table), but enable vertical partitioning of classes. POIDs 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). In this case, the

use of POIDs may incur high update overhead as component objects are deleted or change ownership.

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 into 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* (DSM) 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, 1990]. It can capture object access locality and is therefore potentially superior when access patterns are well-known. The major difficulty, however, is to clustering an object whose parent has been deleted.

In a distributed system, both DSM and NSM are straightforward using horizontal partitioning. Goblin [Kersten et al., 1994] 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, 1994] 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.

15.5.0.2 Distributed Garbage Collection

An advantage of object-based systems is that objects can refer to other objects using object identifier. 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 and become unreachable (at which time, the object is garbage). In reference counting, a problem can arise where two objects only refer to each other but not referred to by anyone else; in this case, the two objects are basically unreachable (except from each other) but their reference count has not dropped to zero.

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. Garbage collection algorithms typically avoid the cases where the collector may miss tracing some reachable objects, due to concurrent changes to other parts of the object graph, and may erroneously reclaim them. On the other hand, although not desirable, it is acceptable to miss reclaiming a garbage and believe that it is alive.

Designing a garbage collection algorithm for object DBMSs is very complex. 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. There have been a number of proposals starting with [Butler, 1987]. More recent work has investigated fault-tolerant garbage collection techniques for transactional persistent systems in centralized [Kolodner and Weihl, 1993; O'Toole et al., 1993] and client-server [Yong et al., 1994; Amsaleg, 1995; Amsaleg et al., 1995] architectures.

Distributed garbage collection, however, is even harder than centralized garbage collection. For scalability and efficiency reasons, a garbage collector for a distributed system combines independent per-site collectors with a global inter-site collector. Coordinating local and global collections is difficult because it requires carefully keeping track of reference exchanges between sites. Keeping track of such exchanges is necessary because an object may be referenced from several sites. In addition, an object located at one site may be referenced from live objects at remote sites, but not by any local live object. Such an object must not be reclaimed by the local collector, since it is reachable from the root of a remote site. It is difficult to keep track of inter-site references in a distributed environment where messages can be lost, duplicated or delayed, or where individual sites may crash.

Distributed garbage collectors typically 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. However, several algorithms propose distributed garbage collection solutions based on reference counting [Bevan, 1987; Dickman, 1991]. Each solution makes specific assumptions about the failure model, and is therefore incomplete. A variant of a reference counting collection scheme, called "reference listing" [Plainfossé and Shapiro, 1995], is implemented in Thor [Maheshwari and Liskov, 1994]. This algorithm tolerates server and client failures, but does not address the problem of reclaiming distributed cycles of garbage.

Distributed tracing usually combines independent per-site collectors with a global inter-site collector. The main problem with distributed tracing is synchronizing the distributed (global) garbage detection phase with independent (local) garbage reclamation phases. When local collectors and user programs all operate in parallel, enforcing a global, consistent view of the object graph is impossible, especially in an environment where messages are not received instantaneously, and where communications failures are likely. Therefore, distributed tracing-based garbage collection relies on inconsistent information in order to decide if an object is garbage or not. This inconsistent information makes distributed tracing collector very complex, because the collector tries to accurately track the minimal set of reachable objects to at least eventually reclaim some objects that really are garbage. Ladin and Liskov [1992] propose an algorithm that computes, on a central space, the global graph of remote references. Ferreira and Shapiro [1994] present an algorithm that can reclaim

cycles of garbage that span several disjoint object spaces. Finally, Fessant et al. [1998] present a complete (i.e., both acyclic and cyclic), asynchronous, distributed garbage collector.

15.6 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. Although object models were not initially defined with a full complement of a query language, there is now a declarative query facility, OQL [Cattell et al., 2000], defined as part of the ODMG standard. In the remainder, we use OQL as the basis of our discussion. As we did earlier with SQL, we will take liberties with the language syntax.

Although there has been significant amount of work on object query processing and optimization, these have primarily focused on centralized systems. Almost all object query processors and optimizers that have been proposed to date use techniques developed for relational systems. Consequently, it is possible to claim that distributed object query processing and optimization techniques require the extension of centralized object query processing and optimization with the distribution approaches we discussed in Chapters 7 and 8. In this section, we will provide a brief review of the object query processing and optimization issues and approaches; the extension we refer to remains an open issue.

Although most object query processing proposals are based on their relational counterparts, there are a number of issues that make query processing and optimization more difficult in object DBMSs [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) that 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 and the evaluation of a particular method may involve some heavy computation (e.g., comparing two DNA sequences). 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. 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.

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 15.6.1), object query optimization (Section 15.6.2), and query execution strategies (Section 15.6.3).

15.6.1 Object Query Processor Architectures

As indicated in Chapter 6, 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 *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.

It is possible to make the query optimizer 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). This can be achieved by means of modularization that separates of a number of concerns [Blakeley et al., 1993]. For example, the user query language parsing structures can be separated from the operator graph on which the optimizer operates, allowing the replacement of the user language (i.e., using something other than OQL at the top) or making changes to the optimizer without modifying the parse structures. Similarly, the algebraic operator manipulation (logical optimization, or re-writing) can be separated from the execution algorithms, allowing exploration with alternative methods for implementing algebraic operators. These are extensions that may be achieved by means of well-considered modularization and structuring of the optimizer.

An approach to providing search space extensibility is to consider it as a group of *regions* where each region corresponds to an equivalent family of query expressions that are reachable from each other [Mitchell et al., 1993]. The regions are not necessarily mutually exclusive and differ in the queries they manipulate, the control (search) strategies they use, the query transformation rules they incorporate (e.g., one region may cover transformation rules dealing with simple select queries, while another region may deal with transformations for nested queries), and the optimization objectives they achieve (e.g., one region may have the objective of minimizing a cost function, while another region may attempt to transform queries to some desirable form).

The ultimate extensibility can be achieved by using object-oriented approach to develop the query processor and optimizer. In this case, everything (queries, classes, operators, operator implementations,, meta-information, etc) are all first-class objects [Peters et al., 1993]. The search space, the search strategy and the cost function are modeled as objects. Consequently, using object-oriented techniques, it is easy to add new operators, new re-write rules, or new operator implementations [Özsu et al., 1995b; Lanzelotte and Valduriez, 1991].

15.6.2 Query Processing Issues

As indicated earlier, query processing methodology in object DBMSs is similar to its relational counterpart, but with differences in details as a result of the object model and query model characteristics. In this section we will highlight 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.

15.6.2.1 Algebraic Optimization

Search Space and Transformation Rules.

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 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 subclass 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 $P\sigma_F < Q_1 \dots Q_k >$), where `union` and `intersection` have the usual set-theoretic semantics, and `select` selects objects from one set P using the sets of objects $Q_1 \dots Q_k$ as parameters (in a sense, a generalized form of semijoin). The results of these operators are sets of objects as well. It is, of course, possible to specify the usual set-theoretic, syntactic rewrite rules for these operators as we discussed in Chapter 7.

What is more interesting is that the relationships mentioned above allow us to define semantic rules that depend on the object model and the query model. Consider the following rules 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):

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

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 Özsu, 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.

Enumerative algorithms based on dynamic programming with various optimizations are typically used for search [Selinger et al., 1979; Lee et al., 1988; Graefe and McKenna, 1993]. 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 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 15.6.2.2, 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* (that we introduced in Chapters 7 and 8) 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. Unfortunately, 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 due to encapsulation. If the query optimizer is considered "special" and allowed to look at the data structures used to implement objects, 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]. In systems that uniformly implement everything as first-class objects, the cost of an operator can be a method defined on an operator implemented as a function of (a) the execution algorithm and (b) the collection over which they operate. In both cases, more abstract cost functions for operators are 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.

15.6.2.2 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 15.4.2 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 have been developed to traverse path expressions forward and backward [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,b].

Rewriting and Algebraic Optimization.

Consider again the path expression we used earlier: 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 [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. A similar attempt to optimizing path expressions within an algebraic framework has been devised based on joins, using an operator called *implicit join* [Lanzelotte and Valduriez, 1991]. 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 15.6.3 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.

15.6.3 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.

15.6.3.1 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 creates an index on each class traversed by a path [Maier and Stein, 1986; Bertino and Kim, 1989]. In addition to indexes on path expressions, it is possible to define indexes on objects across their type 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.

15.6.3.2 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 a multi-way join needs to be computed. 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.

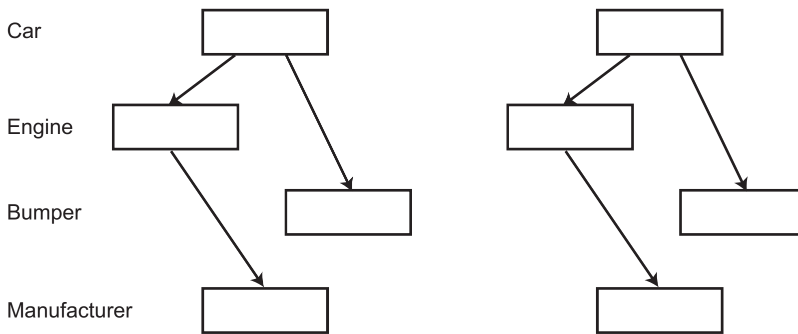

Fig. 15.3 Two Assembled Complex Objects

Example 15.8. Consider the example given in Figure 15.3, which assembles a set of Car objects. 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 15.4a). The assembly operator begins by choosing among the current outstanding references, say $C1$. After resolving (fetching) $C1$, two new unresolved references are added to the list (Figure 15.4b). Resolving $C2$ results in two more references added to the list (Figure 15.4c), and so on until the first complex object is assembled (Figure 15.4g). 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 15.4h). ♦

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

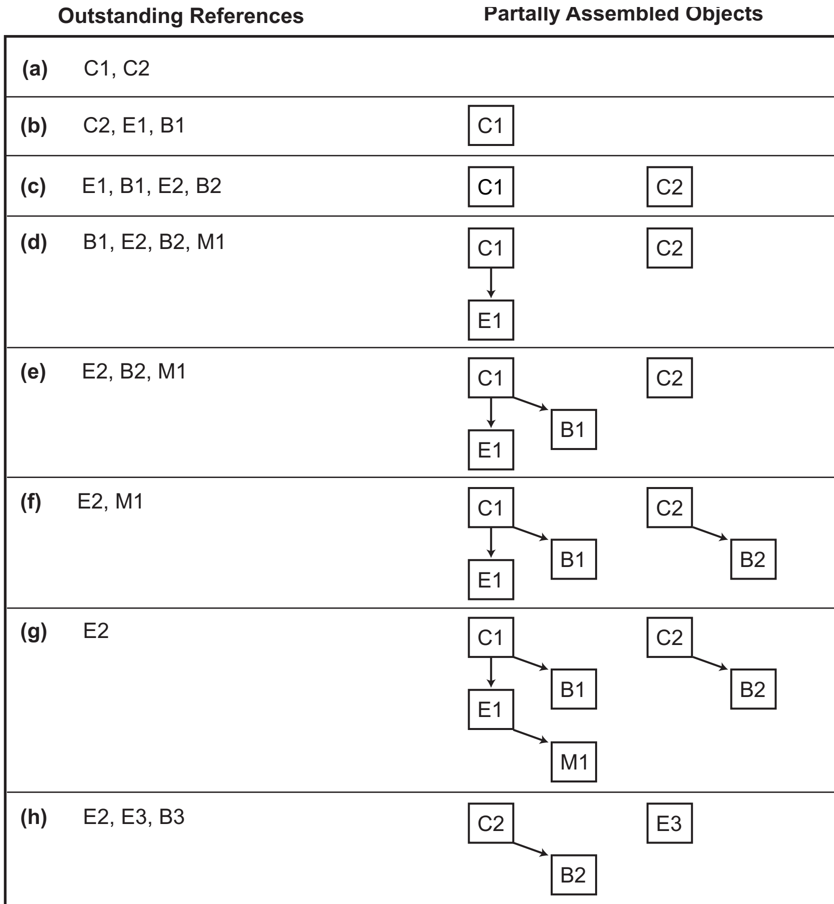

Fig. 15.4 An Assembly Example

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 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.

15.7 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 that clearly requires more work. In this section we will briefly discuss 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., 1982; 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.

15.7.1 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 are based on how a *conflict* is defined. We will concentrate on three alternatives: *commutativity* [Weihl, 1988, 1989; Fekete et al., 1989], *invalidation* [Herlihy, 1990], and *recoverability* [Badrinath and Ramamritham, 1987].

15.7.1.1 Commutativity

Commutativity states that two operations conflict if the results of different serial executions of these operations are not equivalent. We had briefly introduced commutativity within the context of ordered-shared locks in Chapter 11 (see Figure 11.8). 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., 1982].

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, 1989]) makes use of the semantics of operations and their termination conditions.

Example 15.9. 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[7]. ♦

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 15.10. In Example 15.7, 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. ♦

15.7.1.2 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.

[7] 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).

15.7.1.3 Recoverability

Recoverability [Badrinath and Ramamritham, 1987] is another conflict relation that has been defined to determine serializable histories[8]. 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.

15.7.2 Transaction Models and Object Structures

In Chapter 10, we considered a number of transaction models ranging from flat transactions to workflow systems. All of these alternatives access simple database objects (sets of tuples or a physical page). In the case of object databases, however, the database objects are not simple; they can be objects with state and properties, they can be complex objects, or even active objects (i.e., objects that are capable of responding to events by triggering the execution of actions when certain conditions are satisfied). The complications added by the complexity of objects is significant, as we highlight in subsequent sections.

15.7.3 Transactions Management in Object DBMSs

Transaction management techniques that are developed for object DBMSs need to take into consideration the complications we discussed earlier: they need to employ more sophisticated correctness criteria that take into account method semantics, they need to consider the object structure, they need to be cognizant of the composition and inheritance relationships. 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.

---

[8] 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].

15.7.3.1 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 [Hadzilacos and Hadzilacos, 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 that 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 is multigranularity locking [Garza and Kim, 1988; Cart and Ferrie, 1990]. Multigranularity locking defines a hierarchy of lockable database granules (thus the name "granularity hierarchy") as depicted in Figure 15.5. 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.

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

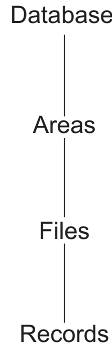

Fig. 15.5 Multiple Granularities

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 15.6.

A possible granularity hierarchy is shown in Figure 15.7. The lock modes that are supported and their compatibilities are exactly those given in Figure 15.6. 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	-	-	+	-	-

Fig. 15.6 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.

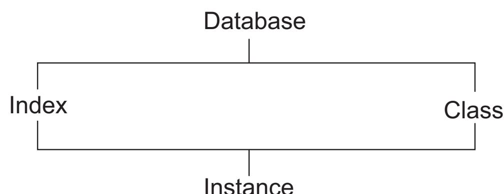

Fig. 15.7 Granularity Hierarchy

- 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.

15.7.3.2 Management of Class Lattice

One of the important requirements of object DBMSs is dynamic schema evolution. 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.

One way to deal with these two problems is, again, by using multigranularity locking. 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. This can be

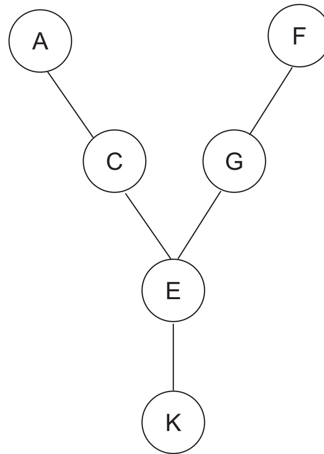

Fig. 15.8 An Example Class Lattice

handled by placing *explicit* locks, rather than implicit ones, on subclasses. Consider the type lattice of Figure 15.8, 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.

An alternative to setting explicit locks is to set 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 Weihl'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 list 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 Figure 11.8). 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_j or its descendants.

15.7.3.3 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.

One approach is based on multigranularity locking where one can 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. An 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

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

Fig. 15.9 Compatibility Matrix for Composite Objects

a composite object. The compatibility of these modes is shown in Figure 15.9. 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.

Another approach extends multigranularity locking by replacing the single static lock graph with a hierarchy of graphs associated with each type and query [Herrmann et al., 1990]. There is a "general lock graph" that controls the entire process (Figure 15.10). 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 that 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 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).

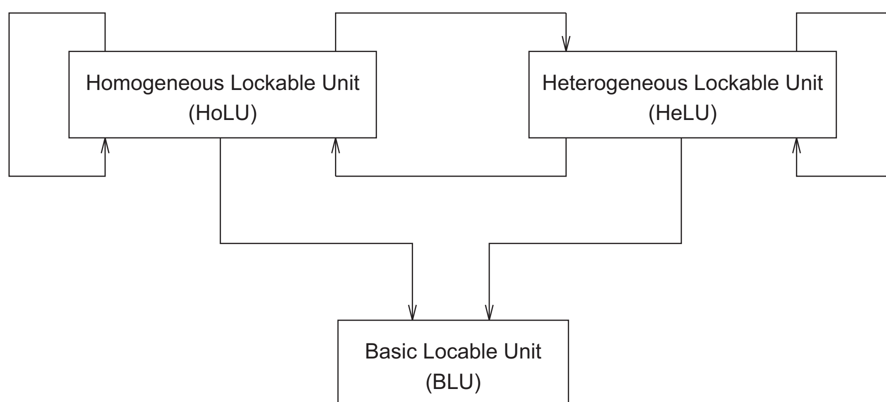

Fig. 15.10 General Lock Graph

Badrinath and Ramamritham [1987] 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*; and
- *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 due to Muth et al. [1993] has the following distinguishing characteristics:

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[9].
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; and
- 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 by 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 that invoke other methods 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.

---

[9] 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].

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.

15.7.4 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. The arguments are as follows:

1. In object DBMSs, what is stored are not only data, but operations on data (which are called methods, behaviors, operations in various object models). Queries that access an object database refer to these operations as part of their predicates. In other words, the execution of these queries invokes various operations defined on the classes (types). To guarantee the safety of the query expressions, existing query processing approaches restrict these operations to be side-effect free, in effect disallowing them to update the database. This is a severe restriction that should be relaxed by the incorporation of update semantics into the query safety definitions.
2. As we discussed in Section 15.7.3, transactions in object DBMSs affect the class (type) lattices. Thus, there is a direct relationship between dynamic schema evolution and transaction management. Many of the techniques that we discussed employ locking on this lattice to accommodate these changes. However, locks (even multi-granularity locks) severely restrict concurrency. A definition of what it means to update a database, and a definition of conflicts based on this definition of update semantics, would allow more concurrency. It is interesting to note again the relationship between changes to the class (type) lattice and query processing. In the absence of a clear definition of update semantics and its incorporation into the query processing methodology,

most of the current query processors assume that the database schema (i.e., the class (type) lattice) is static during the execution of a query.

3. There are a few object models (e.g., OODAPLEX [Dayal, 1989] and TIGUKAT [Özsu et al., 1995a]) that treat all system entities as objects. Following this approach, it is only natural to model transactions as objects. However, since transactions are basically constructs that change the state of the database, their effects on the database must be clearly specified.
Within this context, it should also be noted that the application domains that require the services of object DBMSs tend to have somewhat different transaction management requirements, both in terms of transaction models and consistency constraints. Modeling transactions as objects enables the application of the well-known object techniques of specialization and subtyping to create various different types of TMSs. This gives the system extensibility.
4. Some of the requirements require rule support and active database capabilities. Rules themselves execute as transactions, which may spawn other transactions. It has been argued that rules should be modeled as objects [Dayal et al., 1988]. If that is the case, then certainly transactions should be modeled as objects too.

As a result of these points, it seems reasonable to argue for an approach to transaction management systems that is quite different from what has been done up to this point. This is a topic of some research potential.

15.8 Conclusion

In this chapter we considered the effect of object technology on database management and focused on the distribution aspects when possible. Research into object technologies was widespread in the 1980's and the first half of 1990's. Interest in the topic died down primarily as a result of two factors. The first was that object DBMSs were claimed to be replacements for relational ones, rather than specialized systems that better fit certain application requirements. The object DBMSs, however, were not able to deliver the performance of relational systems for those applications that really fit the relational model well. Consequently, they were easy targets for the relational proponents, which is the second factor. The relational vendors adopted many of the techniques developed for object DBMSs into their products and released "object-relational DBMSs", as noted earlier, allowing them to claim that there is no reason for a new class of systems. The object extensions to relational DBMSs work with varying degrees of success. They allow the attributes to be structured, allowing non-normalized relations. They are also extensible by enabling the insertion of new data types into the system by means of *data blades*, *cartridges*, or *extenders* (each commercial system uses a different name). However, this extensibility is limited,

as it requires significant effort to write a data blade/cartridge/extender, and their robustness is a considerable issue.

In recent years, there has been a re-emergence of object technology. This is spurred by the recognition of the advantages of these systems in particular applications that are gaining importance. For example, the DOM interface of XML, the Java Data Objects (JDO) API are all object-oriented and they are crucial technologies. JDO has been critically important in resolving the mapping problems between Java Enterprice Edition (J2EE) and relational systems. Object-oriented middleware architectures such as CORBA Siegel [1996] have not been as influential as they could be in their first incarnation, but they have been demonstrated to contribute to database interoperability [Dogac et al., 1998a], and there is continuing work in improving them.

15.9 Bibliographic Notes

There are a number of good books on object DBMSs such as [Kemper and Moerkotte, 1994; Bertino and Martino, 1993; Cattell, 1994] and [Dogac et al., 1994]. An early collection of readings in object DBMSs is [Zdonik and Maier, 1990]. In addition, object DBMS concepts are discussed in [Kim and Lochovsky, 1989; Kim, 1994]. These are, unfortunately, somewhat dated. [Orfali et al., 1996] is considered the classical book on distributed objects, but the emphasis is mostly on the distributed object platforms (CORBA and COM), not on the fundamental DBMS functionality. Considerable work has been done on the formalization of object models, some of which are discussed in [Abadi and Cardelli, 1996; Maier, 1986; Chen and Warren, 1989; Kifer and Wu, 1993; Kifer et al., 1995; Abiteboul and Kanellakis, 1998b; Guerrini et al., 1998].

Our discussion of the architectural issues is mostly based on [Özsu et al., 1994a] but largely extended. The object distribution design issues are discussed in significant more detail in [Ezeife and Barker, 1995], [Bellatreche et al., 2000a], and [Bellatreche et al., 2000b]. A formal model for distributed objects is given in [Abiteboul and dos Santos, 1995]. The query processing and optimization section is based on [Özsu and Blakeley, 1994] and the transaction management issues are from [Özsu, 1994]. Related work on indexing techniques for query optimization have been discussed in [Bertino et al., 1997; Kim and Lochovsky, 1989]. Several techniques for distributed garbage collection have been classified in a survey article by Plainfossé and Shapiro [1995]. These sources contain more detail than can be covered in one chapter. Object-relational DBMSs are discussed in detail in [Stonebraker and Brown, 1999] and [Date and Darwen, 1998].

Exercises

Problem 15.1. Explain the mechanisms used to support encapsulation in distributed object DBMSs. In particular:

(a) Describe how the encapsulation is hidden from the end users when both the objects and the methods are distributed.
(b) How does a distributed object DBMS present a single global schema to end users? How is this different from supporting fragmentation transparency in relational database systems?

Problem 15.2. List the new data distribution problems that arise in object DBMSs, that are not present in relational DBMSs, with respect to fragmentation, migration and replication.

Problem 15.3 (**). Partitioning of object databases has the premise of reducing the irrelevant data access for user applications. Develop a cost model to execute queries on unpartitioned object databases, and horizontally or vertically partitioned object databases. Use your cost model to illustrate the scenarios under which partitioning does in fact reduce the irrelevant data access.

Problem 15.4. Show the relationship between clustering and partitioning. Illustrate how clustering can deteriorate/improve the performance of queries on a partitioned object database system.

Problem 15.5. Why do client-server object DBMSs primarily employ data shipping architecture while relational DBMSs emply function shipping?

Problem 15.6. Discuss the strengths and weaknesses of page and object servers with respect to data transfer, buffer management, cache consistency, and pointer swizzling mechanims.

Problem 15.7. What is the difference between caching information at the clients and data replication?

Problem 15.8 (*). A new class of applications that object DBMSs support are interactive and deal with large objects (e.g., interactive multimedia systems). Which one of the cache consistency algorithms presented in this chapter are suitable for this class of applications operating across wide area networks?

Problem 15.9 (**). Hardware and software pointer swizzling mechanisms have complementary strengths and weaknesses. Propose a hybrid pointer swizzling mechanism that incorporates the strengths of both.

Problem 15.10 (**). Explain how derived horizontal fragmentation can be exploited to facilitate efficient path queries in distributed object DBMSs. Give examples.

Problem 15.11 (**). Give some heuristics that an object DBMS query optimizer that accepts OQL queries may use to determine how to decompose a query so that parts can be function shipped and other parts have to be executed at the originating client by data shipping.

Problem 15.12 (**). Three alternative ways of performing *distributed* complex object assembly are discussed in this chapter. Give an algorithm for the alternative where complex operations, such as joins and complete assembly of remote objects, are performed at remote sites and the partial results are shipped to the central site for final assembly.

Problem 15.13 (*). Consider the airline reservation example of Chapter 10. Define a Reservation class (type) and give the forward and backward commutativity matrixes for it.