5. ODBMS Standards

Many different groups are engaged in developing standards that use object concepts. These include:

- application-specific standards groups, e.g.
  — CAD Framework Initiative (electrical CAD)
  — PDES/STEP (mechanical CAD)
  — PCTE (CASE)
  — OSI/NMF (telephony; network management)
  — ANSI X3H6 (CASE)
  — ANSI X3H4 (IRDS)

- groups working on generic object standards
  — ISO ODP, ANSI X3T3 (Open Distributed Processing)
  — ANSI X3H7 (Object Information Management)
  — ANSI X3T5.4 (managed objects)

- object-oriented programming language groups, e.g.
  — ANSI X3J20 (Smalltalk)
  — ANSI X3J16 (C++)

Object concepts are becoming pervasive in standards (the above list is illustrative only, and by no means exhaustive), and many standards that did not previously include object facilities are being extended to include them. Examples include the standards for COBOL, PASCAL, and SQL.

The number and range of interest of these groups (as well as the details of the ways their object models or use of object concepts differ) illustrate the enormous scope of the introduction of object concepts into standards. All of these are potentially relevant to object DBMSs, because ODBMSs may be used in many different applications, and hence may be required to store objects conforming to many different object models.

Of particular interest in the context of ODBMSs are activities directed specifically toward standards governing the use of object concepts in databases, or in distributed object architectures in which object databases might be components. Standards are a key issue in ODBMSs, since:

- the lack of an ODBMS standard is one of the factors that is frequently cited as having restrained the more general use of ODBMSs.

- the lack of a standard integrating both ODBMS and relational DBMS capabilities discourages the use of ODBMSs in conjunction with existing relational systems (themselves largely based on the SQL standard).

This section describes the key standards activities in this area. Specifically, Section 5.1 describes the proposed standards for ODBMSs developed by the Object Database Management Group (ODMG), an industry consortium of OODBMS vendors. Section 5.2 describes the work by ANSI and ISO standardization committees on object extensions to SQL (known as "SQL3"). Section 5.3 describes the standards for distributed object architectures being developed by the Object Management Group (OMG), an industry consortium of developers and users of object-oriented products. Finally, Section 5.4 compares and contrasts these standards, and discusses some issues relating to integrating them.

5.1 ODMG-93

ODMG-93 [Cat94a; Atw93,94] is a proposed set of standards for ODBMSs prepared by the Object Data Management Group (ODMG). The ODMG is a consortium of OODBMS vendors (originally, Object Design, Objectivity, Ontos, O_2, and Versant), and is now affiliated with OMG. ODMG is thus not an accredited standards body in the sense of the ISO or ANSI standards committees. However, the ODMG is likely to have considerable influence on the direction of ODBMS standards. In addition, the OMG is very active in its liaison activities with relevant accredited standards groups. The OMG has adopted a Persistence Service which endorses ODMG-93 as one of several standard interfaces for storing persistent object state (see Section 5.3). The current voting member companies (Object Design, POET, Servio, O_2, Versant, Objectivity, ONTOS), and most of the reviewer member companies (Hewlett Packard, Persistence, Itasca, ADB, Texas Instruments), are committed to support ODMG-93 by 1995.

This section can only provide a brief overview of the concepts in ODMG-93, and is largely taken from [Cat94a], which should be consulted for a more complete explanation of the concepts. [Cat94b] also contains an excellent overview of both the ODMG and the other standards considered in this section (as well as much of the other material in this report). Both books are easily obtainable in (or through) technical bookstores.

ODMG-93 includes the following components:

- A common *Object Model* to be supported by ODBMSs. The OMG Object Model was used as the basis for this object model. In accordance with the OMG "Core + Components" object model architecture (see Section 5.3), an ODBMS *profile* is included that specifies the additional *components* which, together with the OMG model, make up the ODMG model.

- An *Object Definition Language* (ODL). This corresponds to the data definition language used in older database systems. Its syntax is based on that of OMG's Interface Definition Language (IDL).

- An *Object Query Language* (OQL). This is a declarative (nonprocedural) language for querying and updating database objects. It is based in many respects on the relational standard SQL, with variations required to support the more powerful capabilities of an object-oriented language. [Cat94a] notes: "We have not used the extended relational standard in progress, SQL3, because of limitations in its data model and because of its historical 'baggage'. However, we hope that OQL and SQL3 can converge at a future date."

- A *C++ language binding*. This includes a C++ *Object Manipulation Language* (OML), which defines how to write portable C++ code that manipulates persistent objects. The C++ binding also includes a version of the ODL that uses C++ syntax, a mechanism to invoke OQL, and procedures for operations on databases and transactions.

- A *Smalltalk language binding*. This provides capabilities similar to the C++ language binding, but tailored for Smalltalk. Both language bindings are designed to allow reading and writing the same database from both

Smalltalk and C++, provided that a common subset of supported data types is used. [Cat94a] notes that additional chapters may be added later for other language bindings.

The specification also includes a description of how ODBMSs can be integrated into a distributed object system designed according to OMG's CORBA specifications [OMG91], using a special Object Database Adaptor component. This would allow objects stored in an ODBMS to participate as CORBA objects, and also allow ODBMS objects to access CORBA objects on the network.

5.1.1 Object Model

The ODMG object model is similar to those used in a number of the OODBMSs described in Section 3. The basic modeling primitive is the *object*. Objects are categorized into *types*. All objects of a given type exhibit common behavior and a common range of states. The behavior of objects is defined by a set of *operations* that can be executed by an object of the type. An operation is specified by its *signature*: the operation name, argument names and types, possible exceptions, and result types. All operations are defined on a single object type (even though they may take objects of other types as parameters); operations cannot be defined independently of an object type, or on two or more object types (i.e., the ODMG is a *classical* object model, as opposed to a *generalized* object model [Man94]).

The state of objects is defined by the values they have for a set of *properties*. Properties consist of either *attributes* of the object itself, or *relationships* between the object and one or more other objects. Attributes are a kind of property defined on a single object type. Attributes take literals as their values. Attributes are accessed by get_value and set_value operations; they are defined as part of the type interface - there are no implications about the implementation of the object type. Attributes are not first class objects (they cannot have properties or be subtyped), however the built-in get_ and set_value operations can be overridden. (Note that because attributes take literals as their values, defining, for example, a `Department` "attribute" of a `Person` object having a `Department` object as its value requires the use of an ODMG *relationship* instead.) Relationships are a kind of property defined between two mutable object types. Relationships are not objects themselves. A relationship can be one-to-one, one-to-many or many-to-many. Relationships are defined in the interface(s) of the object type(s) involved in the relationship as a "traversal path" from one type to another. A two-way relationship will have a name in each type interface, and an inverse" clause in each path declaration.

The following example ODL object type interface definitions, taken from [Cat94a], illustrate these concepts (keywords are in Boldface).

```
interface Course

// type properties:
(  extent courses
   keys name, number)

// instance properties:
{
   attribute String name;
   attribute String number;
   relationship List<Section> has_sections
         inverse Section::is_section_of
```

```
            {order_by Section::number};
    relationship Set<Course> has_prerequisites
            inverse Course::is_prerequisite_for;
    relationship Set<Course> is_prerequisite_for
            inverse Course::has_prerequisites;

// instance operations:

    Boolean offer (in Integer semester) raises (already_offered);
    Boolean drop (in Integer semester) raises (not_offered);
};

interface Employee
(   extent employees
    key (name, id))

{
    attribute String name;
    attribute Short id;
    attribute Unsigned Short annual_salary;
    void hire();
    void fire() raises (no_such_employee);
};

interface Professor: Employee
(   extent professors)
{
    attribute Enum Rank {full, associate, assistant} rank;
    relationship Set<Section> teaches
            inverse Section::is_taught_by;
    Short grant_tenure() raises (ineligible_for_tenure);
};

interface TA: Employee, Student
()
{
    relationship Section assists inverse Section::has_TA;
};
```

A type has one *interface*, and one or more *implementations*. The interface defines the externally-visible properties and operations supported by all instances of the type. An implementation defines the physical representation of instances of the type, and the *methods* that implement the operations defined at the interface. The ODL defines only object interfaces. The C++ and Smalltalk language bindings provide language-compatible means for defining both object interfaces and implementations.

Types are themselves objects, and thus may have properties. The interface definition of a type specifies values for these type properties. Three type properties are illustrated in the above examples, supertypes, extents, and keys. Object types may be defined as having *supertypes* (e.g., `Professor` is defined as having `Employee` as a supertype). All of the attributes, relationships, and operations defined for a supertype are inherited by the subtype. The subtype may add additional properties and operations, and may also refine the properties and operations it inherits to specialize them to the behavior and range of state values appropriate for instances of the subtype. A type may have more than one supertype, as illustrated by type `TA`, which has `Employee` and `Student` as supertypes. The model defines type inheritance only (i.e., subtyping); it does not define implementation inheritance. An object type also has an *extent*, the set of all instances of the type. The type definer may request that the system automatically maintain a current index to the members of this set by including an **extent** declaration in the type definition (as in type `Course`) naming the set (`courses`). The maintenance of this extent is optional. A type may also be defined as having one or more *keys*. These are properties or sets of properties whose values uniquely identify type instances. For example, type `Course` is uniquely identified by the value of either the `name` or `number` property. Note that a key may be either an attribute or a relationship.

The ODMG object model defines a set of built-in types. The subtype/supertype relationships among this set of types are shown in Figure 5.1.1, taken from [Cat94a]. The types shown in plain text can have instances. The types shown in boldface italics are *abstract* types, meaning that they define characteristics that can be inherited by other types, but cannot have direct instances themselves.

```
Denotable_Object
   Object
         Atomic_Object
               Type
               Exception
               Iterator
         Structured_Object
               Collection  <T>
                     Set<T>
                     Bag<T>
                     List<T>
                           String
                           Bit_String
                     Array<T>
               Structure<e1:T1...en:Tn>
   Literal
         Atomic_Literal
               Integer
               Float
               Character
               Boolean
         Structured_Literal
               Immutable_Collection<T>
                     Immutable_Set<T>
                     Immutable_Bag<T>
                     Immutable_List<T>
                           Immutable_String
                           Immutable_Bit_String
                     Immutable_Array<T>
                     Enumeration
               Immutable_Structure<e1:T1...en:Tn>
                     Date
                     Time
                     Timestamp
                     Interval
Characteristic
   Property
         Attribute
         Relationship
   Operation
```

Figure 5.1.1. ODMG-93 Built-In Type Hierarchy

All objects are of type Denotable_Object. An object can be mutable (instance of type Object) or immutable (instance of type Literal). All denotable objects have *identity*, but the representation used to maintain that identity is different for objects and literals. The representation of the identity of a literal is typically the bit pattern that encodes its value. The representation of the identity of an object is an *object identifier* (OID), a unique value generated specifically to uniquely identify the object.

Instances of type Object have the built-in properties has_name?: Boolean, names: Set<String>, and type: Type, and the built-in operations delete() and same_as? (oid: Object_id) -> b: Boolean. Built-in properties and operations for other predefined types are also specified.

Objects and literals can be atomic or structured. Structured objects (aggregates) can be of type Structure or type Collection. Structures are records with named slots which can be filled by objects or literals of different types. Collections are homogeneous (modulo subtyping) groupings of elements that may or may not be ordered. Structured objects are type templates (or type generators) that take one or more types (T) as parameters. The collection types have operations that support clustering (using pragmas), predicate-based selection, cursor-based operations using iterators, and optional index creation and deletion operations.

All the types shown in Figure 5.1.1 are instances of type Type. Type Type itself is both a subtype and an instance of type Atomic_Object. The metadata that defines a database schema is available via a predefined schema, and can be accessed using the standard DML.

The object model is strongly typed. Every denotable object has a type, and every operation requires typed operands. Two (denotable) objects have the same type if and only if they have been declared to be instances of the same named type. Objects that have been declared to be instances of two different types are not of the same type, even if the types define the same set of properties and operations. If TS is defined as a subtype of T, then an object of type TS can be used in place of an object of type T, but not the reverse. Rules are also specified for type compatibility among literals.

Object lifetime is orthogonal to type, is specified at object creation and, once specified, cannot be changed. Lifetime can be "coterminus with procedure" meaning it is declared in the heading of a procedure, allocated out of the stack and returned to the free pool when the procedure ends, "coterminus with process" meaning it is allocated by the programming language runtime, or "coterminus with database" meaning it is managed by the DBMS runtime. Lifetime is not applicable to literals (immutable objects). Literals always exist implicitly. Most queries return literals.

ODMG-93 defines a nested transaction model. The model uses a standard lock-based approach for concurrency control, and supports a traditional pessimistic concurrency control model (but it does not preclude systems from supporting additional concurrency control policies). A built-in type Transaction is defined, which supports the operations begin, commit, abort, checkpoint, and abort_to_top_level (for nested transactions). Each database is defined as an instance of the built-in type Database. Type Database supports the operations open, close, contains_object?, and lookup_object. Type Database may also be extended by implementors to support various database administration operations such as move, copy, backup, etc.

5.1.2 Object Definition Language and Object Query Language

The ODMG-93 Object Definition Language (ODL) is an extension of OMG's CORBA IDL and, like IDL, is intended to be programming-language independent. The C++ and Smalltalk bindings specify bindings of the ODL to the syntax of these particular languages. ODL is not intended to be a complete programming language, but rather to be a way of specifying interface signatures. In particular, ODL defines only the signatures of object operations, and does not address the definition of the methods that implement those operations. These are to be specified in various programming languages. The example interface definitions given in the last section illustrate the general characteristics of ODL.

The ODMG-93 Object Query Language (OQL) provides facilities for specifying queries on database schemas defined using ODL. OQL is based on the object query language research behind the O_2 ODBMS (see Section 3.7). It is not intended to be computationally-complete, although queries can invoke object methods, and methods written in any of the supported programming languages can include queries. An abstract syntax is provided for OQL, together with several concrete syntaxes: an SQL-like syntax, and syntaxes for using query language expressions in both C++ and Smalltalk. OQL does not provide explicit update operators, such as the UPDATE, INSERT, and DELETE operators in SQL. Instead, OQL allows update operations that have been defined for individual object types (e.g., the `fire()` operation defined for type `Employee` above) to be invoked within query expressions. (One of the things that distinguishes an object model from the relational model is the ability of objects to define their own update operations, rather than having to rely on a fixed set of operations defined by the object model or its query language).

OQL is strongly-typed, and supports queries on denotable objects (as defined the type hierarchy in Figure 5.1.1) that can return atomic or structured objects, and atomic or structured literals. Query expressions effectively define operations whose results have specific types that can be inferred from the expression itself. For example, the query [Cat94a]:

```
select distinct x.age
from x in Persons
where x.name = "Pat"
```

selects the set of ages of persons named "Pat". This query is defined as returning a literal of type `Set<integer>`. `Persons` is not a type, but is rather the extent (set of all instances) of type `Person`. The query

```
select distinct struct(a: x.age, s: x.sex)
from x in Persons
where x.name = "Pat"
```

is similar, but returns, for each person, a structure containing the person's age and sex. The result is a literal of type `Set<struct>`. The query

```
select distinct struct(name: x.name, hps: (select y
                                           from y in subordinates
                                           where y.salary>100000))
from x in Employees
```

builds a still more complex structure. For each employee, the query returns a structure with the employee's name and a set of the employee's highly-paid subordinates. In this case, the result of the query is a literal of type `set<struct(name: string, hps: bag<Employee>)>`. The `from` clause in such queries can reference multiple collections, assigning a separate range variable to each, as in:

```
select x.age, y.location
from x in Employees, y in Departments
where <some qualifying expression>
```

These collections are (conceptually) formed into a Cartesian product prior to evaluating the `where` clause, just as in SQL.

Queries can also reference names assigned to objects (as described above, a set of names is a built-in property of all objects). For example, if a particular employee has been designated as Chairman by assigning the name `Chairman` to the employee's object, the name itself can be used as a simple query, i.e., `Chairman` returns the employee object designated as Chairman. Similarly, `Chairman.subordinates` returns the set of subordinates of the Chairman, and `Persons` returns the set of all persons.

New objects can be constructed by specifying the type name of the object to be constructed, together with values (which may be queries) for the properties of the new object. For example,

```
Employee(name: "Peter", boss: Chairman)
```

creates a new `Employee` object.

Aggregate operations, set operations (union, etc.), and operations for lists, are also defined, as well as operations for flattening nested structures. For example,

```
group x in Employees
by (low: x.salary < 1000,
    medium: x.salary >= 1000 and x.salary < 10000,
    high: x.salary >= 10000)
```

returns a set of three elements, each of which has a property called `partition` which contains the set of employees in the defined category. (Note that `group` in OQL is a separate query, rather than a clause within a query as GROUP BY is in SQL. Section 5.1.4 discusses this further.) The type of the result is:

```
set<struct(low: boolean, medium: boolean, high: boolean,
           partition: set<Employee>)>
```

Examples such as this one illustrate the flexibility in constructing object structures as query results provided by OQL.

5.1.3 Language Bindings

ODMG-93 defines language bindings for both C++ and Smalltalk. These bindings define the "tight" sort of programming language interface generally found in OODBMSs, i.e., a direct mapping of ODMG object model concepts to programming language object model concepts, support for both persistent and transient instances of programming language types, more-or-less transparent movement of objects between the database and application program, etc.

The ODMG-93 C++ binding of ODL is expressed as a class library and an extension to the standard C++ class definition grammar. The class library provides classes and functions to implement the concepts defined in the ODMG object model. The C++ OML syntax and semantics are those of standard C++ in the context of this class library.

Figure 5.1.2 (taken from [Cat94a]) shows the intended usage of the components of the ODMG C++ binding. This is consistent with the way in which many of the C++-based ODBMSs described in previous sections handle the elements of their C++ implementations. (The specification also describes a possible future extension of this binding process that includes a preprocessor capable of processing method bodies as well as class declarations.)

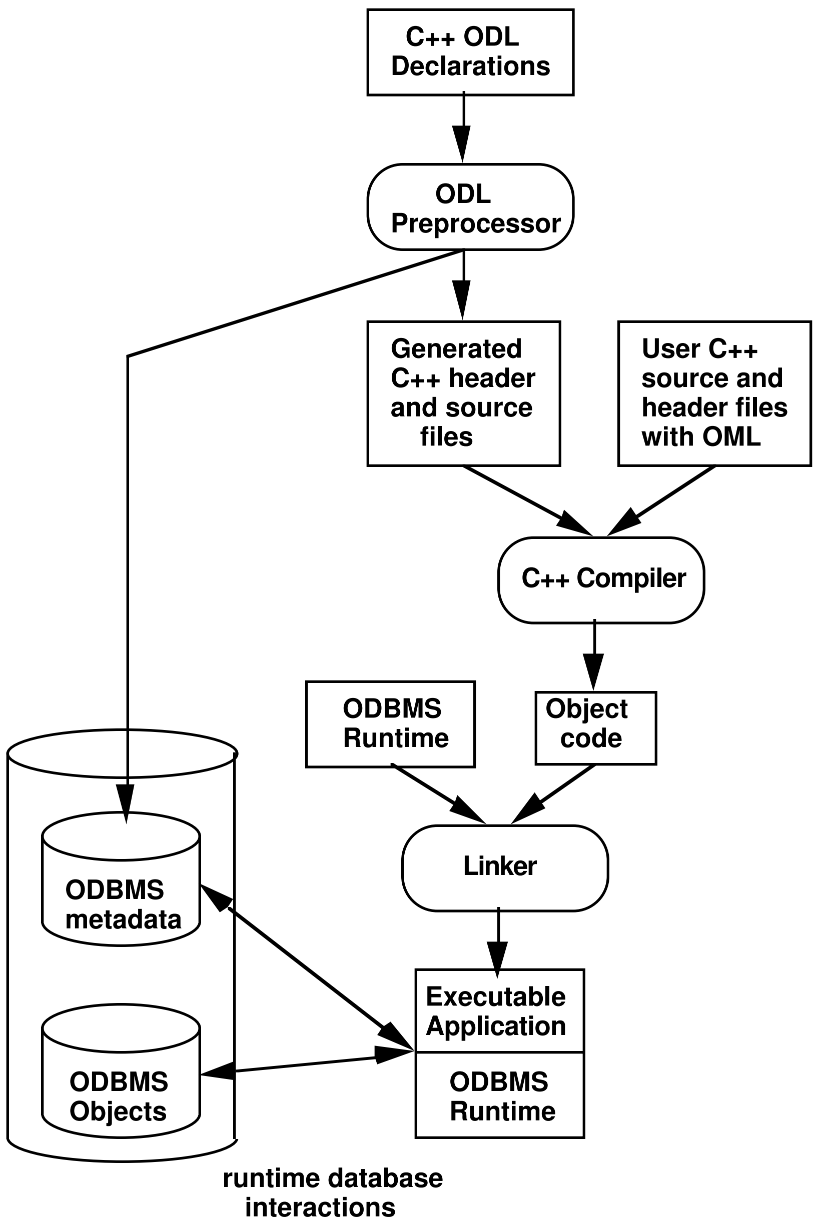

Figure 5.1.2. C++ ODL/OML Processing

The C++ binding is based on a "smart pointer" or "Ref-based" approach. In this approach, the C++ binding maps the ODMG object model into C++ by defining a set of classes, called "persistence-capable classes", that can have both persistent and transient instances. These classes are distinct from normal C++ classes. The C++ binding defines class `Persistent_Object` as the superclass of all persistence-capable objects. Instances of classes derived from `Persistent_Object` can be either persistent or transient. (Some implementations, although they accept the `Persistent_Object` superclass specification as an indication of potential persistence, need not physically introduce a `Persistent_Object` class). In addition, for each persistence-capable class T, another (smart pointer) class `Ref<T>` is also defined (there is also a `Ref<Any>` class defined that provides for generic references to any type). Instances of persistence-capable classes are referenced using these parameterized reference classes. For example,

```
Ref<Professor> profP;
```

declares the variable `profP` as having type `Ref<Professor>`, and

```
Ref<Department> deptRef;
```

declares the variable `deptRef` as having type `Ref<Department>`. Statement

```
profP->grant_tenure();
```

invokes the `grant_tenure` operation defined for class `Professor` on the object referred to by `profP`, and

```
deptRef = profP->dept;
```

assigns the value of the `dept` attribute of the professor referred to by `profP` to the variable `deptRef`.

An example of a class definition using the C++ binding is:

```
class Professor : public Persistent_Object {
public:
// properties:
   int                 age;
   int                 id_number;
   String                    office_number;
   String                    name;
   Ref<Department>     dept         inverse professors;
   Set<Ref<Student>>   advisees     inverse Student::advisor;
//operations:
   void                grant_tenure();
   void                assign_course(Course &);
private:
   ...
};
```

The keyword `inverse` is shown in bold to indicate that it is not defined by C++. As the example shows, C++ already includes the notion of dividing a class definition into two parts: its interface (public part) and its implementation (protected and private members and function definitions). However, in C++, only one implementation is possible for a given class.

Instances of persistence-capable classes may contain embedded members of C++ built-in types, user-defined classes, or pointers to transient data. Applications can refer to these embedded members using C++ pointers (*) or references (&) only during the execution of a transaction. When a transaction is committed, ordinary (non-Ref) pointers and Refs to transient objects are set to the value 0 by the commit operation.

All accesses to persistent objects are made using methods defined on the classes `Ref`, `Persistent_Object`, and `Database`. The dereference operator -> is used to access the members of the persistent object referred to by a given object reference. How an object reference is converted to a C++ pointer is implementation-defined. A dereference operation on an object reference always guarantees that the object referred to is returned (if the object exists but is not in memory, it is automatically retrieved from disk, mapped into memory, and returned), or an error is raised (if the referenced object does not exist).

Persistent objects that have been destroyed or modified must communicate to the runtime ODBMS process that their states have changed. The ODBMS will then update the database with these new states when the enclosing transaction commits. Object change is communicated by invoking a `mark_modified` member function defined for persistence-capable classes (i.e., for class `Persistent_Object`) on the changed object, as in

```
obj_ref->mark_modified()
```

This call is supposed to be included in any methods that modify persistent objects.

The C++ binding also includes a mapping of OQL semantics to C++. Queries can be specified from within a C++ program either by using a `query` method defined for all `Collection` classes, or by using a free-standing `oql` function that is not associated with any class.

The ODMG-93 Smalltalk binding is still evolving. However, the general principles are described in [Cat94a] as a basis for further work. Figure 5.1.3, taken from [Cat94a], shows the intended usage of the components of the ODMG Smalltalk binding.

The basic idea is that an ODL compiler would process ODL declarations and generate both database metaobjects and corresponding Smalltalk class objects. The Smalltalk class objects would be generated so as to automatically support aspects of the ODMG object model. For example, Smalltalk has no built-in concept of attributes. Thus, ODL attribute declarations would be mapped to Smalltalk classes containing pairs of get and set methods for the ODL attributes.

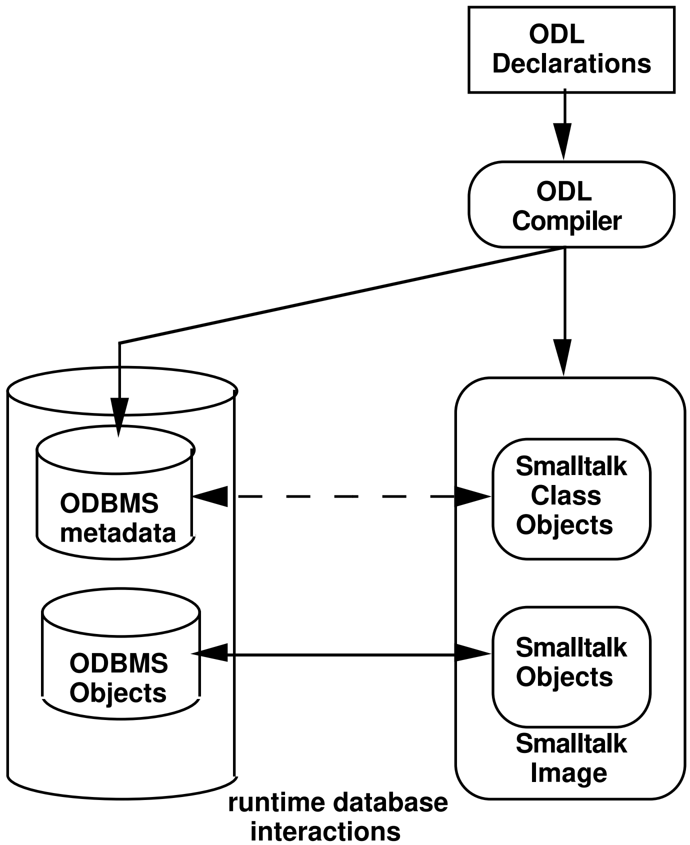

Figure 5.1.3. ODMG Smalltalk Processing

The Smalltalk OML consists of a set of methods added to the Smalltalk classes `Object` and `Behavior`, plus a new class `Session`. In the Smalltalk binding, any Smalltalk class is persistence-capable. An object is made persistent by sending it a `persist` message. A transient object that participates in relationships with a persistent object will become persistent when a transaction commit occurs (this approach is called *transitive persistence* or persistence by *reachability*, and is consistent with Smalltalk reference semantics). As in conventional Smalltalk, there is no notion of explicit deletion of objects. An object is deleted (from the Smalltalk image and the database) during garbage collection when the object is no longer referenced by another object. A persistent object is automatically swapped into the Smalltalk image whenever it is sent a message. A persistent object in the image can be modified using the normal Smalltalk mechanisms. Modified objects have their updated values automatically reflected in the ODBMS when the enclosing transaction commits. This may require the use of an explicit `markDirty` message, as in the C++ binding (and in the current VERSANT Smalltalk interface), depending on the particular Smalltalk implementation involved. A `Session` object is used to manage the connection with a database. A Smalltalk application must open a Session with a database before any objects in that database can be accessed. Transactions are implemented using methods defined on class `Session`. OQL operations are implemented using the Smalltalk methods defined on collections.

[Cat94a] notes that additional work is required to integrate object languages (C++, Smalltalk, CLOS, etc.) to allow them to operate on a multi-language shared object space. Examples of things to be reconciled include:

- garbage collection (as used in Smalltalk) with explicit deletion (as used in C++)
- single-object persistence (as used in C++) with transitive persistence (as used in Smalltalk)
- the different class hierarchies used in the various languages

5.1.4 Remarks

The ODMG object model provides a rich set of type constructors for defining both immutable objects ("literals") and mutable objects ("objects"). Both immutable and mutable types can be either atomic or structured. The result is that application entities can for the most part be modeled either using literal or object types. For example, a collection of application entities can be modeled as a table (multiset or bag) of rows, just as in the relational model, or as multiset of objects. In addition, mutable and immutable types can be freely mixed (e.g., an object type can have a structured literal type as one of its attributes; the structured literal type can have an object type as one of its components; and so on). The OQL query language defined in the ODMG-93 specifications is capable of querying any of these structures.

Overall, the ODMG specifications are an excellent start toward the ultimate development of standards for Object DBMSs. Moreover, they are generally well-integrated with the various relevant OMG specifications (e.g., CORBA, Object Services). This basis in work by OMG means that many of the issues of interoperability between ODMG-based ODBMSs and other aspects of a distributed object architecture can be addressed in a straightforward way. The OQL work in particular provides a solid basis for strongly-typed object query languages that provide the general power of SQL, but with the addition of object facilities.

At the same time, there remain issues to be addressed by further work on the ODMG specifications. For example, the ODMG specifications currently contain a number of ambiguities and minor technical nits that should be corrected. Some of these will be identified in subsequent discussion. Also, a number of facilities provided in SQL, such as views, are not currently provided. Finally, as mentioned at the beginning of this section, convergence with SQL3 is an issue to be addressed in the future. [Cat94a] identifies a number of facilities to be added in later work on the specifications. The ODMG is currently working on its next major revision, ODMG-95. Some of the issues that may be addressed in this and later revisions are discussed in Section 5.4, which compares and contrasts the various standards covered in Section 5, and discusses a possible convergence path between ODMG and SQL3.

In two articles [Kim94a,b], Won Kim, President of UniSQL, has commented on the features of ODMG-93. Since these are the first independent published comments on ODMG-93, and have been widely circulated, we comment on them here. Kim notes as positive aspects of ODMG-93 that the specifications represent more concrete progress than "the current confused state of SQL3", and that they provide a good specification of compound data types and facilities for handling them. He also applauds the introduction of strong typing in the OQL.

Kim also noted a number of criticisms. We note the individual criticisms below, together with some comments.

1. ODMG-93 insufficiently deals with queries that involve methods or nested data: only one simple example of a query involving use of a method, and one simple example with a path expression, with no additional syntactic or semantic specification. These semantics are too complex to be treated in this simple manner. In particular:

- It should be possible to allow a method anywhere in a search condition where an attribute name may be used, but implementation considerations make this difficult (unsafe methods are possible, query optimization is difficult, methods may be either on a client or a server). These considerations are not reflected in the specifications.

- Path expressions are complicated if a compound attribute is cited in the path. Also, again, citing methods involves implementation considerations that are not considered.

Comment: The semantics of path expressions and method invocation certainly require additional description in ODMG-93. The ODMG specifications note the need to provide additional information for query optimization, and suggest the use of pragmas to do this. Additional information of this type could be useful (examples can be found in the Illustra ORDBMS described in Section 4.2); however, a great deal of such information would be implementation-dependent. This is also a problem that ODMG shares with SQL3, because both specifications support user-defined operations.

2. "The OQL allows a query on only a single type, but does not allow an inheritance-hierarchy query (i.e. a query that is targeted to an entire type hierarchy)." (I.e., in a type hierarchy having Employee as a subtype of Person, sometimes the intention is to query persons who are not employees, sometimes only employees, and sometimes all persons (including instances of the Employee subtype of Person).

Comment: First, queries in OQL are targeted to *collections*, which may or may not be extents of types, and not to types themselves. [Cat94a] notes that an object that is an instance of a type is a member of the type extent, and that the extent of a subtype is a subset of the extent of its supertype. In other words, an extent of a type is the collection of all instances in the subtype hierarchy of that type. (Although this does not apply to the point about querying an inheritance hierarchy, OQL queries can be targeted to multiple such collections, since the `from` clause can reference multiple collections, which are then used to form a Cartesian product, just as in SQL). What needs clarification in the ODMG model is how to specify a collection of all objects of exactly a given type and no others (i.e. not including objects of subtypes). Although type Object has a "type" property, it is not specified how comparison works for that property (if x is of specific type `Employee`, and `Employee` is a subtype of `Person`, is `x.type = Person` true or false?) Having both a "is_of_type" and an "is_of_compatible_type" predicate could be used to deal with this issue (see Section 5.4 for further discussion).

3. ODMG-93 proposes a different syntax and terminology from SQL for facilities whose semantics are identical or compatible with SQL. Examples cited are: DIFFERENCE in SQL is EXCEPT in OQL; GROUP BY HAVING in SQL is GROUP BY WITH in OQL; ORDER BY in SQL is SORT BY in OQL;

Comment: In the case of DIFFERENCE vs. EXCEPT, there is an inconsistency in the ODMG specifications. The OQL uses EXCEPT, while the object model defines a "difference" operation for sets and bags. However, contrary to what Kim suggests, the term for set difference in the SQL standard is EXCEPT, rather than DIFFERENCE [Mel94, MS93].

Regarding the other operations, these are not, strictly speaking, just syntactic differences in terminology. These operations are named differently because they are separate operations and not additional clauses for a SELECT query. For example, GROUP BY WITH is a stand-alone query in OQL, and the WITH clause is actually used to define a computation over groups (it is not a HAVING clause, which is a WHERE for groups). In SQL such a computation would be defined as an aggregate operation in the SELECT clause. This is possible for ODMG because the resulting nested structures are definable in the ODMG object model, whereas, since they are not relations, they cannot be treated as separate structures in SQL.

Although one could write queries that result in grouping or sorting as defined in SQL (e.g., GROUP x in Select(etc.) BY etc.), having Group and Sort as separate operations allow them to be combined in interesting ways with other operations. For example,

```
Select struct(E: e, Proj: Group p in e.Projects by (Mgr: p.mgr))
From e in Employees
Where e.location = "NY"
```

returns type `set< struct(E: employee, Proj: set< struct(Mgr: manager, partition: set<project>) >) >`.

4. ODMG-93 is missing major features from SQL such as views, authorization, triggers, and dynamic schema changes (add and drop methods, add and drop types/subtypes/supertypes).

Comment: This is true. However, Object DBMSs often include such operations, and these implementations would presumably be the basis for rapidly adding such facilities.

5. There are no SQL-like statements (insert, delete, create) for creating new objects, updating a set of objects, etc., based on search conditions.

Comment: The OQL specification explicitly notes that it relies on type-defined operations for doing updates. This is possible because such operations can be defined (and in fact *must* be defined) for object types, rather than defining general UPDATE operations for tables, as in SQL. Creation operations are also defined for instances of types. Operations for inserting and removing members of sets are also defined. The use of these operations in performing general types of updates should, however, be more thoroughly explored in ODMG specifications (or related explanatory material). Discussion within the SQL community has noted that this is an issue that also must be resolved in SQL3's object facilities (since one must somehow be able to invoke ADT operations that perform updates within SQL3).

5. ODMG-93 defines a nested transaction model as the standard. Kim says that flat transactions should be the default, with nested transactions as an option.

Comment: This is debatable. It seems as if one could define an SQL binding to ODMG that maps to only the flat subset of ODMG-93. Also, given the extended applications to which the SQL3 object extensions might be applied, it might be a better approach for SQL3 to add a nested transaction model (these are supported by many transaction monitors that might be used in SQL environments).

6. Various facilities in ODMG-93 can create a "meta-data nightmare". Examples are:

- allowing multiple names for an object
- specifying the lifetime of a single object
- the ability to create multiple subsets of type extents, such that an object can belong to multiple subsets
- the ability to create an index on a subset of a type extent (i.e., on an arbitrary collection of objects)
- the ability to name query statements and use them in other queries
- the need to name the extent of a type

Comment: This is also debatable. These features are inherent in most object models, and ODMG-93 is at least consistent in including them. They are also available in many ODBMS products. All of these facilities can also be simulated in relations, and it is not clear that there is an obvious reason to prohibit them (in fact, the ability to name specific objects could be highly useful in SQL3). The same tuple can already exist in multiple tables in SQL. The ability to name query statements is provided in SQL through the ability to define SQL statements as procedures, and invoke them in other queries. If users do not wish to create and keep track of different names, collections of objects, etc., in most cases they are free to not use these facilities.

7. In ODMG-93 both the C++ and Smalltalk bindings require all objects to be persistent. Also, these bindings need to address the issue of storing persistent objects that reference nonpersistent objects.

Comment: ODMG-93 release 1.1 [Cat94a] does not appear to require that all objects be persistent. It also addresses the issue of storing persistent objects that reference nonpersistent ones (although some may disagree with how this is done).

8. An overall criticism is that the ODMG object model is a superset of the relational model; therefore, the ODMG query language should be a superset of SQL.

Comment: It is not necessarily true that since the ODMG model subsumes the relational model, the language should subsume SQL. Some of the features of SQL are defined as they are because the underlying capabilities are not directly available in the relational model, as they would be in an object model (see, e.g., the comments on point #3 above). However, it should be possible to map an SQL3 query to something supported by ODMG-93 (in [Cat94c], Rick Cattell, the ODMG Chair, suggests that an SQL3 language binding could be made to ODMG-93). Thus, it might be possible to have some "special case" ODMG syntax that applies to the "relational subset" of the ODMG model.

5.2 SQL3

ANSI (X3H2) and ISO (ISO/IEC JTC1/SC21/WG3) SQL standardization committees have for some time been adding features to the SQL specification to support object-oriented data management. The current version of SQL in progress including these extensions is often referred to as "SQL3". The current SQL3 specification includes the capability to support user-defined abstract data types (ADTs), including methods, object identifiers, subtypes and inheritance, polymorphism, and integration with external languages. Enhancements have also been made to the facilities for defining tables (relations) in SQL3, including row types and row identifiers, and an inheritance mechanism. Additional facilities include control structures and parameterized types to make SQL a computationally complete language for creating, managing, and querying persistent objects. The added facilities are intended to be upward compatible with the current SQL92 standard [MS93, DD94].

The following description is only an overview, and is taken from [EKMS+94; GS91; Gal94; KCS94; Kel94a,b,c; Kul93; Mel94]. It is also based on our interpretation of certain aspects of [Mel94], and hence may contain some misunderstandings. In addition, SQL3 is still very much an evolving specification; some parts are changing rapidly, and a considerable number of issues must still be resolved. Hence this description must be regarded as primarily a "snapshot" of work in progress, with some indications of possible further directions.

SQL3 object extensions are being pursued through extensions to both the *type* facilities and the *table* facilities in SQL. Currently, these extensions overlap somewhat, and are not as thoroughly integrated as they will be after further work (currently ongoing) by the SQL standards committees. The major extension to the type facilities, the abstract data type (ADT) facility, is reviewed first, followed by the extensions to the table facilities (some of which also involve type extensions).

5.2.1 Abstract Data Types

One of the basic ideas behind the object extensions is that, in addition to the normal built-in types defined by SQL, user-defined *abstract data types* (ADTs) may also be defined. These types may be used in the same way as built-in types. For example, columns in relational tables may be defined as taking values of user-defined types, as well as built-in types.

An ADT definition encapsulates attributes and operations in a single entity. In SQL3, an abstract data type (ADT) is defined by specifying a set of declarations of the stored attributes that represent the value of the ADT, the operations that define the equality and ordering relationships of the ADT, and the operations and derived attributes that represent the behavior of the ADT. The operations and derived attributes are implemented by procedures called *routines*.

More precisely, an ADT specification contains:

- the name of the ADT
- an ordering specification, which specifies either a RELATIVE, HASH, EQUALS, or LESS THAN function
- the names of any CAST operations defined for the ADT (these are used to cast the ADT value as the value of some other type)
- the descriptor of each attribute of the ADT
- an indication of whether the ADT is a VALUE or an OBJECT ADT, and if it is an OBJECT ADT, whether it is WITH OID VISIBLE or WITH OID NOT VISIBLE
- the descriptor of each operation that has the ADT as a parameter or result

An example ADT declaration (slightly modified) from [Kul93] is:

```
CREATE OBJECT TYPE person_type
 (name VARCHAR NOT NULL,
  sex CONSTANT CHAR (1),
  age UPDATABLE VIRTUAL GET WITH age SET WITH set_age,
 PRIVATE
  birthdate DATE CHECK (birthdate < > DATE '1992-01-01',
```

```
 PUBLIC
  EQUALS DEFAULT,
  LESS THAN NONE,
  ACTOR FUNCTION age (:P person_type) RETURNS REAL
   RETURN <code to calculate the age>
  END FUNCTION,
  ACTOR FUNCTION set_age (:P person_type, ...) RETURNS person_type
   <code to update the birthdate>
   RETURN :P
  END FUNCTION,
  DESTRUCTOR FUNCTION remove_person (:P person_type)
              RETURNS person_type
  <various cleanup actions>
  DESTROY :P;
  RETURN :P;
  END FUNCTION;
);
```

A (rather simplified) version of the syntax for defining an ADT is:

```
<ADT definition> ::=
        CREATE { VALUE | OBJECT } TYPE <ADT name>
             [ <OID options> ]
             [ <subtype clause> ]
             [ [ CONSTANT | UPDATABLE ] [ <member defns> ] ]

<OID options> ::=
        WITH OID [ [ NOT ] ] VISIBLE

<subtype clause> ::=  UNDER <supertype ADT name list>

<member defn> ::=
        <attribute definition>
      | <routine declaration>
      | <ordering definition>
      | <cast definition>

<attribute definition> ::=  <stored attribute> | <virtual attribute>

<stored attribute> ::=
        [ <encapsulation level> ]
        <attribute name> <data type> }
        [ <constraints and other subclauses> ]

<virtual attribute> ::=
        [ <encapsulation level> ]
        <attribute name>
                [ READ ONLY | CONSTANT | UPDATABLE ] <data type>
                VIRTUAL [GET WITH <routine name> ]
                        [SET WITH <routine name> ]
        [ <constraints and other subclauses> ]

<encapsulation level> ::= PRIVATE | PROTECTED | PUBLIC

<operator name list> ::= OPERATORS <specific routine designator>...

<ordering definition ::=
```

```
                    EQUALS <routine name> | STATE | OID
                  | LESS THAN <routine name> | NONE
                  | RELATIVE <routine name>
                  | HASH <routine name>

<cast definition> ::=  CAST ( <datatype> AS <datatype> ) WITH
                       <routine name>
<routine declaration> ::=  <encapsulation level> <routine>
```

There are two kinds of ADTs, OBJECT ADTs and VALUE ADTs. The key characteristic of an object ADT is that each instance of the ADT has an OID, or object identifier, which is a value that uniquely identifies or designates the ADT instance. The association of an OID value with its corresponding instance is made by making the OID value an implicit attribute of the ADT, having attribute name "OID". The OID of an ADT instance is conceptually separate from the value or state of the instance (the *value* of an ADT instance is defined as the ordered sequence of stored components of the instance, excluding the OID component). OIDs are themselves instances of separate object identifier types; there is a distinct OID type for each ADT type. The OID type is used only to define the implicitly-defined OID attribute in object ADTs. Testing OIDS for equality serves to test for the identity of the corresponding ADT instance. An OID literal actually exists for an object identifier type only if the associated ADT is defined WITH OID VISIBLE. In this case, the OID value is materialized as a character string with an implementation-defined length. If WITH OID NOT VISIBLE is specified, then the OID value may not be passed as a parameter to functions or stored in a host language variable. If the ADT is a VALUE ADT, then the ADT does not have an object identifier; instead, each instance represents itself just like values of primitive data types do.

Each component (attribute or routine) of an ADT has an encapsulation level of either PUBLIC, PRIVATE, or PROTECTED. PUBLIC components form the interface of the ADT and are visible to all authorized users of the ADT. PRIVATE components are totally encapsulated, and are visible only within the definition of the ADT that contains them. PROTECTED components are partially encapsulated; they are visible both within their own ADT and within the definition of all subtypes of the ADT. An attribute A of an ADT instance X may be referred to from within the ADT definition (or from within a routine having access to the ADT's PRIVATE components) using the special notation `X..A` (for example, in the definition of the `person_type` ADT above, a statement in the `age` actor function would refer to the `birthdate` attribute as `:P..birthdate`).

There are two types of ADT attributes, *stored attributes* and *virtual attributes*. A *stored* attribute is specified by giving an attribute name and a data type. The data type of a stored attribute can be any known data type, including another ADT. As noted above, each ADT instance has a value consisting of the collection of values of all stored attributes of that instance. Each stored attribute implicitly declares a pair of functions to get and set the attribute value. The function to get the attribute value has the same name as that of the attribute, while the function to set the attribute value has a name formed by prefixing *set_* to the attribute name. A *virtual* attribute has a value that is derived or computed by a user-defined get_attribute function (specified using the `GET WITH` declaration above). Attributes can be designated as UPDATABLE, READ ONLY, or CONSTANT. Values of updatable attributes can be changed at any time through assignment. Read-only attributes cannot be updated. Constant attributes can be assigned values only at instance creation. The specification of a set_attribute function for a virtual attribute is required only if the attribute is to be UPDATABLE.

Routines (procedures and functions) that define aspects of the behavior of the ADT may also be encapsulated within the ADT definition (these routines have access to the ADT's PRIVATE attributes; routines may also be defined outside an ADT definition, and need not be associated with ADTs). A number of these ADT routines have predefined names. For example, when an ADT is defined, a constructor function is automatically defined to create new instances of the type. The constructor function has the same name as the type and takes zero arguments. It returns a new instance of the type, whose OID field, if any, is set, and whose attributes are uninitialized. The constructor function is PUBLIC.

An ADT specification allows for the definition of user-specified functions that define the ordering used in the definitions of predicates and the ORDER BY clause, and the equality test used in the GROUP BY clause, HAVING clause, UNIQUE predicate, and DISTINCT. EQUALS returns a Boolean if and only if its two arguments are equal. LESS-THAN returns a Boolean if its first argument is less than the other. The RELATIVE function is invoked with two arguments X and Y to return an integer Z. X is defined to be less than, equal to, or greater than Y if and only if Z is less than zero, equal to zero, or greater than zero, respectively. The HASH function is invoked with an argument X of the ADT to obtain a result of some predefined data type P. The relative ordering of X in the ADT is defined to be the same as the ordering of P in its predefined data type. If no ordering clause is specified for a VALUE ADT, then the ordering is effectively EQUALS STATE. If no ordering clause is specified for an OBJECT ADT, then the ordering is effectively EQUALS OID.

Special routines called CAST functions can be defined to specify how to map an ADT to other existing data types (the CAST capability was introduced in SQL92 for built-in scalar data types). For example, an IMAGE ADT may be CAST to a BIT STRING representation using one of these functions. CAST functions allow the ADT definer to define mappings between an ADT and specific external representations while maintaining the encapsulation of the internal representation.

Other routines associated with ADTs include FUNCTION definitions for type-specific user-defined behavior. The FUNCTION definitions specify the data operations on the ADT and return either BOOLEAN, if the result is to be used as a truth value in a Boolean predicate, or a single value of a defined data type, if the result is to be used as a value specification. Functions may either be SQL functions, completely defined in an SQL schema definition, or external function calls to functions defined in standard programming languages, as described in Section 5.2.2.

An SQL-invoked routine that has an ADT type as one of its parameters or as its return type must either appear in that ADT's body (and it can appear in only one such body) or be named in the <operator name list> of those ADTs. This gives the routine access to the PRIVATE attributes and functions of those ADTs.

ADTs can be specified as the data types of columns in tables, parameters in procedures and functions, attributes in other ADT definitions, variables in compound SQL statements, etc. In the case of OBJECT ADTs, it is also possible to specify whether the ADT reference refers to an actual ADT instance (which includes its OID), or the OID of an ADT instance located elsewhere. Specifically, whenever an OBJECT ADT is used in a type definition, e.g., in a column definition of a table, or in a variable definition of a procedure or function, the ADT type reference consists of the ADT name, followed by an optional *elaboration mode*, represented by the keyword INSTANCE. If the ADT is an OBJECT ADT and the reference does not specify INSTANCE, then the data item (column, variable value, etc.) contains an OID that identifies an instance of the abstract data type. Otherwise, the item contains an actual instance of the ADT. This allows for both the usual *reference* semantics found in many object models, as well as for the specification that an object (instance) is to be *embedded* within another ADT instance, or to physically reside in a column of a table. An ADT instance that physically resides in one column of a table can be referenced from other columns of the same or different tables by storing its OID in those columns.

An ADT instance can exist in any location that an ADT name can be referenced. However, the only way that any ADT instance can be stored persistently in the database is to be stored as the column value of a table. For example, in order to store instances of the `person_type` ADT defined above persistently in a database, a table would have to be created with a column having the ADT as its data type, as in:

```
CREATE TABLE persons
     (  person_data   person_type INSTANCE);
```

There is no facility in SQL3 to name individual instances of an ADT, and to store them persistently in the database using only those names. Similarly, there is no central place that all instances of a given ADT are guaranteed to exist (a built-in type extent), unless the user explicitly creates such a place. Thus, in SQL3 it is not necessarily possible to apply SQL query operations to all instances of a given ADT. The instances must first be stored in one or more tables (as column values).

Syntax is provided to query both stored and virtual attributes of ADTs contained in table columns. For example, suppose the following table is defined using the `person_type` ADT defined above):

```
CREATE TABLE emp
     (  person_data   person_type,
        manager       person_type,
        spouse        person_type INSTANCE,
        ...
     )
```

In this case, a query to find the names of people older than 40 would be[11]:

```
SELECT name(e.person_data)
FROM emp e
WHERE age(e.person_data) > 40;
```

This uses the implicitly-defined get-attribute functions for the `name` and `age` attributes of the `person_type` ADT.

ADT encapsulation rules apply to query statements. For example, because the attribute `birthdate` was declared to be PRIVATE, the following query is invalid:

```
SELECT birthdate(e.person_data)
FROM emp e
WHERE age(e.person_data) > 40;
```

In addition to attributes, actor functions that are associated with an ADT can be used in queries involving that ADT. For example, if there is a function `address` with

---

[11] Several of these examples are from [Kul93], updated to reflect the more recent changes in the draft specifications discussed here.

person_type and date as parameters and a character string as a result, the following query could be written:

```
SELECT address(e.person_data, DATE '1951-12-01')
FROM emp e
WHERE age(e.person_data) > 40;
```

Inter-object relationships can be captured by attributes of ADTs that have ADTs as their values. For example, suppose that the person_type ADT had been defined with an additional attribute school of type school_type, and that school_type had an enrollment attribute. Then the query:

```
SELECT enrollment(school(e.person_data))
FROM emp e
WHERE name(e.person_data) = 'John';
```

retrieves the enrollment of the school that John attended. Note that a join was not required in traversing the inter-object relationship.

SQL92 introduced queries nested within the FROM clause in an SQL expression, as in [DD94]:

```
SELECT AVG (X) AS aq
FROM (SELECT SUM (qty) AS x ...)
```

Such nesting can be combined with the use of ADT expressions to provide greater flexibility in constructing the results of query expressions.

A comment in [Mel94] notes that the distinction mentioned above between columns containing ADT instances (i.e., having types specified as <ADT name> INSTANCE) and columns containing ADT OIDs can create some unexpected behavior when an INSTANCE column is assigned to. For example, in the table definition below, the person_data and manager columns contain the OIDs of person-type objects stored in INSTANCE columns in the same or other relations, while the spouse column contains an embedded instance of person_type. If Dick's spouse is assigned to be Jane's manager, then Dick's spouse will be equal to Jane's manager (Jane's manager will contain the OID of Dick's spouse). However, if Jane's manager is assigned to be Dick's spouse, they will not be equal. Instead, a new ADT value, with a different OID, will be stored in the column. This is because assignment to an INSTANCE column requires the creation of a new object (the same instance cannot be stored in more than one place).

```
CREATE TABLE emp
     (  person_data   person_type,
        manager       person_type,
        spouse        person_type INSTANCE,
        ...
     )
```

Also, suppose the emp table had been defined in the following way (i.e., with three INSTANCE columns instead of only one).

```
CREATE TABLE emp
     (  person_data   person_type INSTANCE,
        manager       person_type INSTANCE,
```

```
        spouse        person_type INSTANCE,
        ...
    )
```

In this case, it would be impossible to independently delete any of the individual objects, or the row (if the row were deleted, all the objects embedded in it would also be deleted; an embedded object cannot be deleted without deleting the row containing it, which in this case would also delete the other embedded objects) [KCS94].

These sorts of anomalies suggest that additional work will be necessary to fully integrate objects into SQL in a way that seems natural to users of other object-oriented programming languages and database systems. Work in this area is currently in progress. For example, [Sha94] is a proposal that associates OIDs with the "site" of an ADT instance (i.e., the variable, column, attribute, or parameter) as opposed to associating the OID with an attribute of an ADT instance. [Kel94c] contains an analysis of various anomalies possible with the current OBJECT ADT facility (such as those cited above), and suggests removing OBJECT ADTs entirely, instead using a combination of VALUE ADTs and row identifiers (see Section 5.2.4 below) to provide the same facilities.

5.2.2 Routines

A *routine* in SQL is basically a subprogram (one that is stored in the database). A routine may be either a FUNCTION, which returns a value, or a PROCEDURE, which does not return a value. A FUNCTION may be either a DESTRUCTOR or an ACTOR. A DESTRUCTOR function destroys ADT instances. An ACTOR function is any other function that reads or updates components of an ADT instance, or accesses any other parameter declared in its parameter list (routines may be associated with tables, as described in Section 5.2.4). A routine is specified by giving its name, its parameters, a RETURNS clause if it is a function, and a body. A parameter in the parameter list consists of a parameter name, its data type, and whether it is IN, OUT, or INOUT. The RETURNS clause specifies the data type of the result returned. Routines are invoked using a functional notation.

Different routines may have the same name. This is referred to as *overloading*, and may be required, for example, to allow a subtype to redefine an operation inherited from a supertype (see Section 5.2.3). The parameter lists of such routines must be sufficiently different to distinguish which of the routines is to be invoked for a given invocation. SQL3 implements what is sometimes known as a *generalized* object model, meaning that the types of *all* arguments of a routine are taken into consideration when determining what routine to invoke, rather than using only a single type specified in the invocation as, for example, in C++ or Smalltalk[12]. As a result, the rules for determining which routine to invoke for a given invocation are fairly complex.

A routine may be either an *SQL routine* or an *external routine*. An SQL routine has a body that is written completely in SQL. An external routine has an externally-provided body written in some standard programming language. If the function is an SQL routine, its body is any SQL statement, including compound statements and control statements. A number of new statement types have been added in SQL3 in order to make SQL computationally-complete enough so that object behavior can be completely specified in SQL. These statements are discussed in Section 5.2.5.

---

[12] A routine embedded in the definition of a particular ADT, or named in that ADT's <operator name list>, has access to the PRIVATE attributes of that ADT.

If the function is an external function, the routine body is an external function reference of the form:

```
EXTERNAL NAME <external function name>
LANGUAGE <language name>
[ NOT ] VARIANT
```

Languages that can be specified include ADA, C, COBOL, FORTRAN, MUMPS, PASCAL, and PLI. A VARIANT routine is one that may return different results when called multiple times with the same input arguments.

5.2.3 ADT Subtypes and Inheritance

An ADT can be defined as a subtype of one or more ADTs by defining it as UNDER those ADTs (multiple inheritance is supported). In this case, the ADT is referred to as a *direct subtype* of the ADTs specified in the UNDER clause, and these ADTs are *direct supertypes*. A type can have more than one subtype and more than one supertype. An instance of a subtype is considered an instance of all of its supertypes. An instance of a subtype can be used wherever an instance of any of its supertypes is expected. If an ADT is defined as a VALUE ADT, then all of its supertypes must be VALUE ADTs, and conversely. Similarly, if an ADT is defined as an OBJECT ADT, then all of its supertypes must be OBJECT ADTs, and conversely.

An example of a subtype declaration from [Kul93] is:

```
CREATE TYPE student_type UNDER person_type
(student_id INTEGER,
 major VARCHAR,
 year INTEGER,
 EQUALS DEFAULT,
 LESS THAN NONE,
 DESTRUCTOR FUNCTION remove_student (:S student_type)
            RETURNS student_type
  BEGIN
   <various cleanup actions>
   DESTROY :S;
   RETURN :S;
  END;
 END FUNCTION;
);
```

Every instance is associated with a "most specific type" that corresponds to the lowest subtype assigned to the instance. At any given time, an instance must have exactly one most specific type (in some cases, multiple inheritance must be used to ensure this is true). The most specific type of an instance need not be a leaf type in the type hierarchy. For example, a type hierarchy might consist of a maximal supertype `person`, with `student` and `employee` as subtypes. `student` might have two direct supertypes `undergrad` and `grad`. An instance may be created with a most specific type of `student`, even though it is not a leaf type in the hierarchy.

A subtype definition has access to the representation of all of its direct supertypes (but only within the ADT definition that defines the subtype of that supertype), but it has no access to the representation of its sibling types. Effectively, components of all direct supertype representations are copied to the subtype's representation with the same name and data type. To avoid name clashes, a subtype can rename selected components of the representation inherited form its direct supertypes.

A subtype can define actor and destructor operations like any other ADT. A subtype can also define operations which have the same name as operations defined for other types, including its supertypes (overriding).

5.2.4 Tables

As noted earlier, SQL3 object extensions are also being pursued through extensions to the *table* facilities in SQL. As in SQL92 (and SQL89), a table is a collection of rows, and a row is a nonempty sequence of values. The CREATE TABLE statement is used to create a table in the database. Tables are defined by specifying the table name, and a sequence of columns. Currently in SQL3, a table can be defined as either a SET table, a MULTISET table, or a LIST table. By default, a table is a MULTISET table (a table is a multiset of rows in SQL92). SET tables and LIST tables share all the properties of MULTISET tables, but have the additional properties that a SET table can contain no duplicate rows, and a LIST table has an ordering defined for the rows. Each table has a data type, which consists of the specification of whether the table is a MULTISET, SET, or LIST table, and the *row type* of the table. The row type of a table is the sequence of (column name, data type) pairs specified in the table definition (a new facility to allow the explicit declaration of row types as explicit SQL data types is described below). The data types of columns can include ADTs as well as built-in types. As noted in Section 5.2.1, the only way that an ADT instance can be stored persistently in the database is to be stored as the column value of a table.

The new *row identifier* facility [Kel94a] (accepted by ANSI but not yet by ISO) allows the DBMS to maintain a unique identifier for each row, and allows applications to make use of this identifier. A row identifier can also be used as a column value and/or foreign key. The facility is specifically intended to help reduce the difference between SQL3 rows and objects in conventional object models.

A row identifier is a data type which is used to identify rows in base tables. The value of a row identifier for a given base table row is equal to itself, and is not equal to the value of the row identifier for any other base table row within the database. Any base table may have a row identifier explicitly defined by specifying `WITH IDENTITY` in the table definition. For every table T for which a row identifier is defined, a new subtype `T IDENTITY` of the row identifier data type is defined (hence, the type of a row identifier indicates the name of the table with which it is associated). Every table which has a row identifier defined has an implicit column, named `IDENTITY`, which contains the unique row identifier value (and subtype) for each row. This column is not included in the results of a SELECT * expression in a query, but can be explicitly named if its value is required. For example, the following table definition specifies that table `person` has an additional column named `IDENTITY` which contains a unique value of type `person IDENTITY` to identify each row. In addition, the `spouse` column is declared as taking values of type `person IDENTITY` (the row identifier values of other rows in the table).

```
CREATE TABLE person WITH IDENTITY
    (name CHAR(20),
     sex  CHAR(1),
     age  INTEGER,
```

```
        spouse person IDENTITY);
```

A row identifier may be passed to an application for use as a handle to the data in the row. This allows a reference to a row to be passed without having to actually copy the row's data, and allows access to the data in specific columns when this is required.

Tables have also been enhanced with a *subtable* facility. A table can be declared as a subtable of one or more supertables (it is then a *direct* subtable of these supertables), using an UNDER clause associated with the table definition. The concept of subtable is completely independent from the ADT subtype concept. For example, the following declarations specify a `person` table and two subtables of `person`, `employee` and `customer`:

```
CREATE TABLE person
    (name CHAR(20),
     sex  CHAR(1),
     age  INTEGER,
     spouse person IDENTITY);

CREATE TABLE employee UNDER person
    (salary FLOAT);

CREATE TABLE customer UNDER person
    (account INTEGER);
```

Note that these are *tables* rather than *ADTs*, and thus they define entirely different representations for `person` and `person` subtype information than the ADT definitions of `person_type` and `student_type` given earlier.

When a subtable is defined, the subtable inherits every column from its supertables, and may also define columns of its own. A maximal supertable (a supertable that is not a subtable of any other table) together with all its subtables (direct and indirect) makes up a *subtable family*. A subtable family must always have exactly one maximal supertable. Any row of a subtable must correspond to exactly one row of each direct supertable. Any row of a supertable corresponds to at most one row of a direct subtable.

Any base table which has a subtable or supertable has a row identifier implicitly defined. The row identifier type for a table with supertables is a subtype of the row identifier type defined for each supertable. For example, in the declarations above, table `person` has a row identifier of type `person IDENTITY` implicitly defined, table `employee` has a row identifier of type `employee IDENTITY` implicitly defined, and type `employee IDENTITY` is a subtype of type `person IDENTITY`.

A value of row identifier type X can be substituted for a value of row identifier type Y if X is the same as Y, or if X is a subtype of Y. This means, for example, that the row identifier of an `employee` row can be assigned to the `spouse` column in the `person` table, even though that column is declared to be type `person IDENTITY` (because type `employee IDENTITY` is a subtype of type `person IDENTITY`).

The rules for the SQL INSERT, DELETE, and UPDATE DML statements are defined in such a way as to keep the rows in the tables of a subtable family consistent with each other, in accordance with the rules described above. Specifically:

- If a row is inserted into a subtable T, then a corresponding row (with the same row identifier, and the same values as any values provided for inherited columns of T) is inserted into each supertable of T, cascading upward in the table hierarchy. If T is a maximal supertable, a row is inserted only into T.

- If a row is updated in a *super*table, then all inherited columns in all corresponding rows of the direct and indirect subtables are correspondingly changed.

- If a row is updated in a *sub*table, then every corresponding row is changed so that their column values match the newly updated values.

- If a row in a table that belongs to a subtable family is deleted, then every corresponding row is also deleted.

The semantics maintained are those of "containment"; a row in a subtable is effectively "contained" in its supertables. This means that, for example, a row could exist for a person in the `person` table without a corresponding row in the `employee` table (if the person is not also an employee). A row for a new employee, not corresponding to any existing person, could be inserted into the `employee` table, and this would automatically create a corresponding row in the `person` table.

Using these facilities, routines can be associated with tables to implement object-like operations on rows, and more specialized routines can be associated with subtables to support polymorphism for those operations. For example, the declarations below define `polygon` and `rectangle` tables, and an overloaded `area` operation (with a different routine for each table)[13].

```
CREATE TABLE polygon
    (xvalues LIST(FLOAT),
     yvalues LIST(FLOAT));

CREATE TABLE rectangle UNDER polygon;

CREATE FUNCTION area (P polygon IDENTITY,
             xs LIST(FLOAT), ys LIST(FLOAT)) RETURNS FLOAT
BEGIN

...<complicated polygon area computation>

END;

CREATE FUNCTION area (R rectangle IDENTITY,
             xs LIST(FLOAT), ys LIST(FLOAT)) RETURNS FLOAT
BEGIN

...<simple rectangle area computation>

END;
```

Note that the row identifier is a parameter of the overloaded operation `area`. Since the row identifier's type identifies the table from which the row comes, the appropriate routine

[13] See Section 5.2.6 for a description of the LIST type constructor used in this example.

can be invoked when the operation `area` is applied to a mixed collection of rows from different tables, as in the query:

```
SELECT area(P, xvalues, yvalues) FROM polygon P
```

(Recall that, since table `polygon` is a supertable of table `rectangle`, it contains both rows corresponding to polygons, having row identifiers of type `polygon IDENTITY`, and rows corresponding to rectangles, having row identifiers of type `rectangle IDENTITY`).

The result is that rows with row identifiers combined with table operations become somewhat like objects having only public attributes (i.e., without encapsulation, except possibly what might be provided by defining a view in conjunction with the table).

SQL3 also supports the concept of *domains*. A domain is a set of permissible values. It is defined in the schema, and used to constrain the set of valid values that can be stored in some location. A domain is specified using a CREATE DOMAIN statement which specifies a data type, an optional constraint to further restrict the set of valid values, and an optional default specification. Domains may be used, for example, in table column specifications (in place of ordinary data types).

The new *row type* facility [EKMS+94] (accepted by both ANSI and ISO) specifies a row as a new kind of data type. A row in a table is an instance of one of these types, and every row of the same table has the same type. The components of row types are called *fields*. Two of these types are equivalent (and assignable) if both have the same number of fields and every pair of fields in the same position have compatible types. The intent is to provide a data type that can represent the types of rows in tables, so that complete rows can be stored in variables, passed as arguments to routines, and returned as return values from function invocations. This facility also allows the definition of domains and columns in base tables based on these row types, as in[14]:

```
CREATE DOMAIN us_address ROW
   (street CHAR(30),
    city  CHAR(20),
    zip ROW (original CHAR (5), plus4 CHAR(4))
    );

CREATE TABLE employees
   (last_name CHAR(20),
    first_name CHAR(20),
    age  INTEGER,
    address us_address);
```

The result is a table (`employees`) that has a column (`address`) with a row as a value (the row values can also be *nested*, as illustrated by the `zip` field of `us_address`). SQL expressions can be applied to these columns, as in:

```
INSERT INTO employees
   VALUES('Manola','Frank',
          ('151 Tremont St.',
           'Boston','MA'('02111','0000'))
          );
```

[14] These examples (in some cases slightly modified) are from [EKMS+94].

A field F of a row R is referred to in expressions using the notation `R..F`, similar to that used to refer to ADT attributes from within an ADT definition, as in:

```
UPDATE employees
SET address..zip..original = '02111'
WHERE (last_name,first_name) = ('Manola','Frank');
```

or

```
BEGIN

   DECLARE pres_city ROW (city CHAR (20), state CHAR(2));

   SELECT (address..city, address..state)
   INTO pres_city
   FROM employees
   WHERE title = 'President';

   UPDATE employees
   SET ...
   WHERE (address..city, address..state) = pres_city;

END
```

The SELECT and UPDATE statements above show how row types allow entire rows to be manipulated as units in statements. Since row types are data types, functions can also be defined on them, as in:

```
CREATE FUNCTION correct_zip(address us_address)
    RETURNS BOOLEAN;
   BEGIN
   <function code>
   END;


SELECT COUNT(*)
FROM employees e
WHERE NOT correct_zip(e.address);
```

[EKMS+94] notes that there are overlaps and interactions between the row type facility and other SQL3 facilities, and has identified a number of potential followup proposals to deal with this situation. For example, one possible proposal would extend CREATE TABLE to allow table definitions to refer to row types, e.g.,

```
CREATE DOMAIN name AS ROW
   (first CHAR(30),
    last  CHAR(20));

CREATE TABLE names OF name;
```

which would be equivalent to:

```
CREATE TABLE names
   (first CHAR(30),
    last  CHAR(20));
```

Another possible proposal would be to allow the definition of *subtypes* of row types.

Through 1993, the SQL3 draft defined an association between table definitions and ADT definitions such that an ADT definition could be used in a table definition, with instances of the ADT considered as rows of the table, and with the PUBLIC attributes of the ADT considered as the visible columns of the table. Using this approach, a table could be defined based on the `person_type` ADT definition as follows:

```
CREATE TABLE people OF TYPE person_type;
```

In this case, the table would get its columns (`name`, `sex`, and `age`) from the definition of the `person` ADT, as defined above. A table and a new type could also be created in a single statement. For example, both type `person_type` and table `people` could be created by the single statement:

```
create table people of new type person_type
(
      <ADT declaration for ADT person_type>
);
```

This approach integrated the table and ADT concepts, by allowing the table to be considered as a collection of ADT instances. The approach also integrated the subtable and ADT subtype capabilities. This approach, sometimes called *Table of ADT*, is reflected in [Kul93], but was deleted from the draft specifications when it became clear that it introduced a number of problems, having to do with the integration of ADT and relational concepts, that did not appear to be easily solvable without serious surgery on the relational characteristics of SQL [MD93]. Some of the more recently introduced facilities, such as row identifiers and row types, appear to reflect to some extent a desire to recapture some integration of table-like and object-like facilities, but in a way that is more naturally based on tables. It remains to be seen how far this integration will go.

5.2.5 Procedural Facilities

Support for SQL routines (routines written completely in SQL) was mentioned in Section 5.2.2. SQL routines are useful because they support procedural encapsulation, and allow complete behavior to be specified within an ADT definition (or for a routine associated with a table) without the need to escape to a procedure written in some other language. In addition, complex behavior can be made available to the host application program via a single call. Some of the additional statements provided for writing SQL routines include [GS91]:

- A DESTROY statement that destroys the existence of an OBJECT ADT instance; it is only allowed in a DESTRUCTOR function.
- An ASSIGNMENT statement that allows the result of an SQL value expression to be assigned to a free standing local variable, a column, or an attribute of an ADT.
- A CALL statement that allows invocation of an SQL procedure.
- A RETURN statement that allows the result of an SQL value expression to be returned as the RETURNS value of the SQL function.

- A CASE statement to allow selection of an execution path based on alternative choices.

- An IF statement with THEN, ELSE, and ELSEIF alternatives to allow selection of an execution path based on the truth value of one or more conditions.

- A LOOP statement, with a WHILE clause, to allow repeated execution of a block of SQL statements based on the continued true result of the <search condition> in the WHILE clause. A LOOP statement is also allowed to have a statement label.

Additional control facilities available include compound statements and exception handling. A compound statement is a statement that allows a collection of SQL statements to be grouped together into a "block". A compound statement may declare its own local variables and specify exception handling for an exception that occurs during execution of any statement in the group. For exception handling, an exception declaration establishes a one-to-one correspondence between an SQLSTATE error condition and a user-defined exception name.

SQL3 also provides a module facility for defining stored procedures. In the existing SQL92 standard, a *module* is a persistent object created by the Module Language. It is a named package of procedures that can be called from an application program, where each procedure consists of exactly one SQL statement. However, there is no requirement that an implementation be able to execute Module Language (the alternative is Embedded SQL), and the resulting persistent module is not stored as part of the SQL schema, and is not reflected in the information schema tables. The SQL3 module facility provides the ability to define persistent modules that are part of the SQL schema, and whose procedures may be called from any SQL statement in the same processing environment.

5.2.6 Other Type Constructors

Type templates (parameterized types) are provided in SQL3 to define families of ADTs [GS91]. The syntax for specifying a parameterized type in SQL3 is very similar to that for specifying a regular ADT, except that the keyword TEMPLATE indicates that the specification is for a parameterized ADT rather than a regular ADT. Each type template definition specifies a set of formal parameters and a body. The parameters specify values or data types that must be provided when the type template is used to define a specific generated type. The body is the same as that of an ordinary ADT definition, except that it can contain the formal parameters of the type template.

A parameter of a type template may be any value of a data type known to the SQL environment. A parameter may also be a reference to an existing data type, rather than a value of that type. For example, it is possible to define either VECTOR(9) or VECTOR(INTEGER). A *generated type* is an ADT resulting from a specification of a type template with a set of actual parameters. Each actual parameter must be a value, or a data type, that can be determined at syntax evaluation time, i.e. usually a literal or a data type name. If the actual parameter is a data type name, then the formal template parameter must specify TYPE.

Nesting of data types is allowed in the definition of a parameterized type. For example, two type templates might be specified as:

```
point(::coord_type)
sequence(::object_type)
```

and a new type generated as `sequence(point(FLOAT))`.

SQL3 also provides a facility for the user to declare that two otherwise equivalent type declarations are to be treated as *distinct* data types. The keyword `DISTINCT` used in an type declaration indicates that the resulting type is to be treated as "distinct" from any other declaration of the same type. For example, if two new types are declared as:

```
CREATE DISTINCT TYPE cartesian_point AS point

CREATE DISTINCT TYPE polar_point AS point
```

any attempt to treat an instance of one type as an instance of the other would result in an error, even though each type has all the methods and attributes of the other.

A number of predefined parameterized collection types are also defined. A collection may be specified as `SET(<type>)`, `MULTISET(<type>)`, or `LIST(<type>)`. (Note that these are *data types*, and are thus different from the SET, MULTISET, and LIST *tables* described earlier.) In each case, the `<type>` parameter can be any legal type, e.g., a predefined type, an ADT, or another collection type. For example `SET(INTEGER)` and `SET(LIST(INTEGER))` would both be valid declarations, as would `SET(movie)` and `SET(LIST(movie))`, where `movie` is some previously defined ADT.

A collection can be used as a simple table in queries. In this case, each element of the collection corresponds to a row in the table. The table is treated as having a single column whose type is defined by the type of the instances of the collection. A collection is a VALUE (has no OID). Since collection types are data types, they must be declared as the types of table columns in order to store instances of collections persistently in the database.

Combining row types with the parameterized collection types described above allows the creation of data types whose values come close to being tables. For example, using the `us_address` row type defined in Section 5.2.4, the type `MULTISET(us_address)` could be defined. A value of this type could then be stored in a column or an ADT attribute, and could be queried using the facility for querying collections as if they were tables. This essentially creates a facility for defining nested tables.

5.2.7 Generic ADT Packages

[GS91] notes that while the SQL3 specification provides facilities for defining ADTs, it does not currently specify standard packages of specific ADTs for various application areas. X3H2, together with other groups, is investigating mechanisms to achieve standardization of particular ADT packages. Such packages would be optional specifications that need not be supported in order to claim conformance to SQL. The main intent of standardization would be to allow applications to use the same ADTs across different application areas, thus promoting interoperability and data sharing, and encouraging performance optimization over a manageable collection of types. [GS91] includes descriptions of the following ADT packages:

- Vector Spaces

- Matrix algebra
- Euclidean geometry
- Geographic regions on Earth surface
- Spatial data types
- Text and document data types
- Computer graphics
- Pictures
- Audio
- Video
- HyperMedia data types
- ASN.1 data type

5.2.8 Language Bindings

The SQL (1992) standard defines language bindings for a number of standard languages. For Embedded SQL (the primary mechanism), SQL statements are embedded in the statements of the host programming languages, proceeded by the words EXEC SQL. Embedded SQL is generally implemented using a preprocessor, which analyzes the embedded SQL statements, creates SQL modules representing to the SQL statements, and replaces the statements with calls to the modules. A key aspect of the individual language bindings is the definitions of correspondences between SQL data types and host language data types. In some cases, these are relatively straightforward; e.g., the SQL `CHARACTER` data type maps to a C `char`. In other cases, the mapping is not so straightforward. For example, SQL92 has a `TIMESTAMP` data type, but standard programming languages do not contain a corresponding built-in type. In these cases, SQL requires the use of a `CAST` function to convert database `TIMESTAMP` data to character data in the program, and vice-versa [MS93]. In SQL92, these type correspondences are defined only at the level of elementary scalar data types. There are no type correspondences defined for structured types, e.g., between a row of an SQL table and a flat record or structure in a programming language (although some such correspondences would be relatively straightforward to define).

There are currently no bindings defined between the SQL3 ADT extensions (or rows with row identifier values) and object classes or types in object-oriented programming languages such as C++ or Smalltalk. [BN94] is an X3H2 working paper that contains some preliminary discussion of such bindings. It notes that defining bindings between user-defined SQL ADTs and host language object classes would involve some form of *registration* of the mapping with the DBMS (similar to the registration concept used in some of the ODBMS C++ interfaces discussed in Section 3 to notify the ODBMS of newly-defined C++ classes). [BN94] also notes the need to map between SQL OIDs and host language pointer or OID types.

[BN94] discusses both C++ and Smalltalk mappings. The mapping of the SQL ADT facility to C++ is more straightforward than the Smalltalk mapping, since a number of the concepts in the SQL ADT facility are based on C++ concepts. For example, like C++, SQL ADTs have multiple encapsulation levels (PUBLIC, PROTECTED, and PRIVATE). Similarly, both type systems support a distinction between a type that contains an instance of an ADT, and a type that contains a reference or pointer to the ADT. A Smalltalk mapping would be somewhat more complicated. For example, Smalltalk supports only public functions and private attributes (instance variables). Smalltalk only supports references to object classes (OIDS); there is no concept of an embedded object.

5.2.9 Remarks

SQL3 to a large extent provides two ways to model application entities: rows in tables, and ADT instances. The current specifications contain considerable overlap between the facilities associated with tables, and those associated with ADTs (the existence of both table and ADT inheritance hierarchies, and both row and object identifiers, are examples). Some of this reflects a hybrid approach toward an ODBMS standard which is in many respects natural. Upward compatibility between SQL3 and previous relational standards is an important consideration for the SQL3 developers. To an extent, these different ways of modeling application entities also parallel the facilities for defining both mutable and immutable objects provided by the ODMG specifications (and recent additions to SQL3 such as the row type facility have improved SQL3's facilities in this regard). However, it does not appear that the SQL3 facilities can be as freely mixed as the ODMG facilities, and a tighter integration of some of the SQL3 facilities would be highly desirable. In addition, the various options currently available for storing and referencing object ADTs do not appear very clean, and a number of other aspects of the specifications appear unclear.

In evaluating these comments, it must be remembered that the SQL3 work on objects is at more of an interim stage than the work of ODMG. The SQL3 object facilities are still very much under development, and changes are regularly taking place. Hence it is likely that a far more polished facility will finally be produced than exists today. Some of the issues that may be addressed in this work, and the relationship of this work to that of ODMG, are discussed further in Section 5.4. Also, the SQL3 work involves many extensions to SQL other than the ones related to object facilities that are the main emphasis in this report. Finally, the SQL3 does not start from an object-oriented approach, and the tight integration of object and relational facilities involves addressing many difficult issues.

It remains to be seen in what timeframe an SQL standard containing any of these object capabilities will appear. Some estimates call for a 1996 completion of these capabilities. However, some participants in the SQL standardization community apparently feel that these capabilities are problematic enough to require delaying their inclusion until after the next official standard, concentrating in the meantime on more "relational" extensions. This could push these facilities into 1997 or 1998.

William Kelley and Won Kim, both of UniSQL, have commented on the features of SQL3 in a working paper [KK94] sent to the X3H2 standards committee. This paper somewhat parallels Won Kim's papers commenting on the ODMG-93 specifications referred to in Section 5.1. The general point in [KK94] is that SQL3's object facilities should more closely resemble "the general object model" (a set of common object-oriented features supported by many object-oriented programming languages, the ODMG specification, and, e.g., by the UniSQL and Illustra ORDBMSs described in Section 4). [KK94] notes a number of specific contrasts between the characteristics of "the general object model" and those of SQL3. For example:

1. SQL requires a single root table for a table hierarchy; i.e., if you have a table C multiply-inheriting from tables A and B, tables A and B must be descended from a common table R. Also, there must be a primary key inherited in these tables, and it must be defined in the root table. The general object model does not impose these requirements.

2. A single table in an SQL3 table hierarchy cannot be a query scope. Instead, the query scope is always the entire hierarchy.

3. It is not possible to determine the table that a row belongs to (e.g., given a row, you cannot tell whether it came from EMPLOYEE or PERSON). This makes it impossible to specialize operations for subtypes (i.e., it is not possible for dispatching to select the implementation of an operation based on the instance being a particular type).

4. Rows in SQL do not have identity, only ADTs with OIDs; rows require primary keys.

5. The general object model does not define inheritance for ADTs; nor does it define "ADT with OID".

Many of these criticisms have a legitimate basis, but the way they are expressed may be confusing. This is particularly so since the criticisms are, for the most part, directed at SQL3's *table* facilities, rather than comparing the "general object model" with SQL3's *object* (ADT) facilities, which might appear to be the most comparable facilities. For example, point #1 above compares table inheritance with object model type inheritance. However, relational tables (at least prior to the adoption of the row identifier facility) were not equivalent to object types; in SQL3 table inheritance they combined the features of types and collections of instances. The SQL3 facility most directly corresponding to inheritance in an object model is ADT inheritance, which generally *does* resemble inheritance in an object model. Similarly, point #3 above compared the lack of any ability in SQL3 to overload operations on tables with the ability an object model provides to overload operations on object subtypes. However, table inheritance in SQL3 (again, prior to the row identifier facility) did not attempt to provide operator overloading on tables. *ADT type inheritance* in SQL3 (which again was the most directly-comparable facility) *does* provide operator overloading on ADT subtypes.

The reason that these criticisms were aimed at the table facilities is apparently because the authors of [KK94] believe that SQL's table facilities should be enhanced in various ways so that they can take on the characteristics of objects, rather than using the object ADT facility for this purpose. This is illustrated by UniSQL proposals introducing facilities such as the row identifier facility (see Section 5.2.4) to address the above criticisms, as well as by a number of other UniSQL proposals, e.g., [Kel94c] which suggests eliminating object ADTs, and using row identifiers and value ADTs instead. As a result of the adoption of some of these proposals, many of the criticisms above have now been addressed, although currently object ADTs remain, and it is not completely clear what direction will be taken to rationalize the various overlapping facilities that now exist in SQL3.

An appropriate subject of discussion would be whether these object extensions have gone far enough, e.g., whether tables should not simply be considered a special case of a collection of object instances, with rows being merely one kind of built-in structured data type that could have instances in such collections. The criticism would then be that tables and ADT collections currently represent alternative ways of representing application abstractions, and that these are not sufficiently integrated.

In other cases, [KK94] asserts characteristics that the authors may *prefer* as being characteristics of the general object model, e.g., that "an OID under the general object model generally consists of the entity-type identifier (i.e., class identifier) as well as the instance identifier". This particular implementation of OIDs is used in some object systems, but not in others. (Also, it is unnecessary that the OID contain type information in order to be able to determine the type of an instance, which may be logical requirement the authors have in mind).

In still other cases, the comments appear to be somewhat debatable. For example, considering point #5, the object types in object models certainly are certainly ADTs, have oids, and most of them define inheritance, although it is certainly true that the specific OBJECT ADT facility in SQL3 has aspects that are rather dubious, and do not resemble those in other object models, as noted in Section 5.2.1. In addition, the type theory literature contains many examples of ADTs without oids, and which define inheritance (or at least subtyping), and the ODMG object model also supports user-defined literal object types (i.e., ADTs without oids), just as SQL3 does.

It should also be observed at this point that no one has really identified a single "general object model". For example, the X3H7 Object Model Features Matrix [Man94] describes many variations on the general object concept. It is certainly true that the most popular object models (e.g., C++ and Smalltalk) have many of the characteristics that [KK94] describes, but even in these models there are many differences in detail that can be significant, some of which will be mentioned in Section 5.4.

However, the addition of facilities such as row identifiers and row types reflects a movement of SQL3 in the direction of trying to address the outstanding difficulties. Hopefully, further work of this sort will result in a more complete specification, with better integration of table and object facilities.

5.3 OMG Standards

The Object Management Group (OMG) is an industry consortium with the goal of developing an object-oriented architecture for integration of distributed applications. OMG emphasizes as goals the interoperability, reusability, and portability of components, and its operating procedures attempt to insure that any specifications adopted as standards have their basis in commercially available software (however, OMG adopts only *interface* standards, not the software itself).

The OMG has published several documents, including:

- Object Management Architecture (OMA) Guide [OMG92a]
- Common Object Request Broker Architecture (CORBA) [OMG91]
- Common Object Services Specification (COSS), Volume I [OMG94a]

The following subsections briefly discuss aspects of each of these specifications.

5.3.1 OMG Object Management Architecture

The OMG's Object Management Architecture (OMA), as described in the OMG Object Management Architecture Guide [OMG92a], defines a Reference Model which identifies and characterizes the components, interfaces, and protocols that compose a distributed object architecture.

Figure 5.3.1, taken from [OMG92a], shows the four main parts of the OMA Reference Model.

- The *Object Request Broker* (ORB) enables objects to make and receive requests and responses in a distributed environment. The ORB and related facilities are defined by the CORBA specification [OMG91].

- *Object Services* is a collection of services (interfaces and objects) that provide basic functions for using and implementing objects.

- *Common Facilities* is a collection of services that provide general purpose capabilities useful in many applications.

- *Application Objects* are objects specific to particular end-user applications.

The Object Request Broker in the OMA is viewed as a "messaging backplane" spanning multiple systems. From the software point of view, the ORB may consist of a single software component, or multiple cooperating (and possibly heterogeneous) software components. The Object Services, Common Facilities, and Application Objects in the OMA correspond to different *categories* of object implementations. In the OMA Reference Model, the purpose of categorizing objects into Object Services, Common Facilities, and Application Objects is to guide OMG's strategy for developing interface specifications. Object Services include lower-level (and thus the most crucial) object interfaces; Common Facilities provide standardized interfaces to common application services; and Application Objects reflect the need for independently-developed application interfaces for which the OMG will never develop specifications.

The OMA also defines a common core object model to which all the objects in the architecture are to conform. The object model plays the role of providing a semantic "umbrella" governing the definitions of the other components of the architecture.

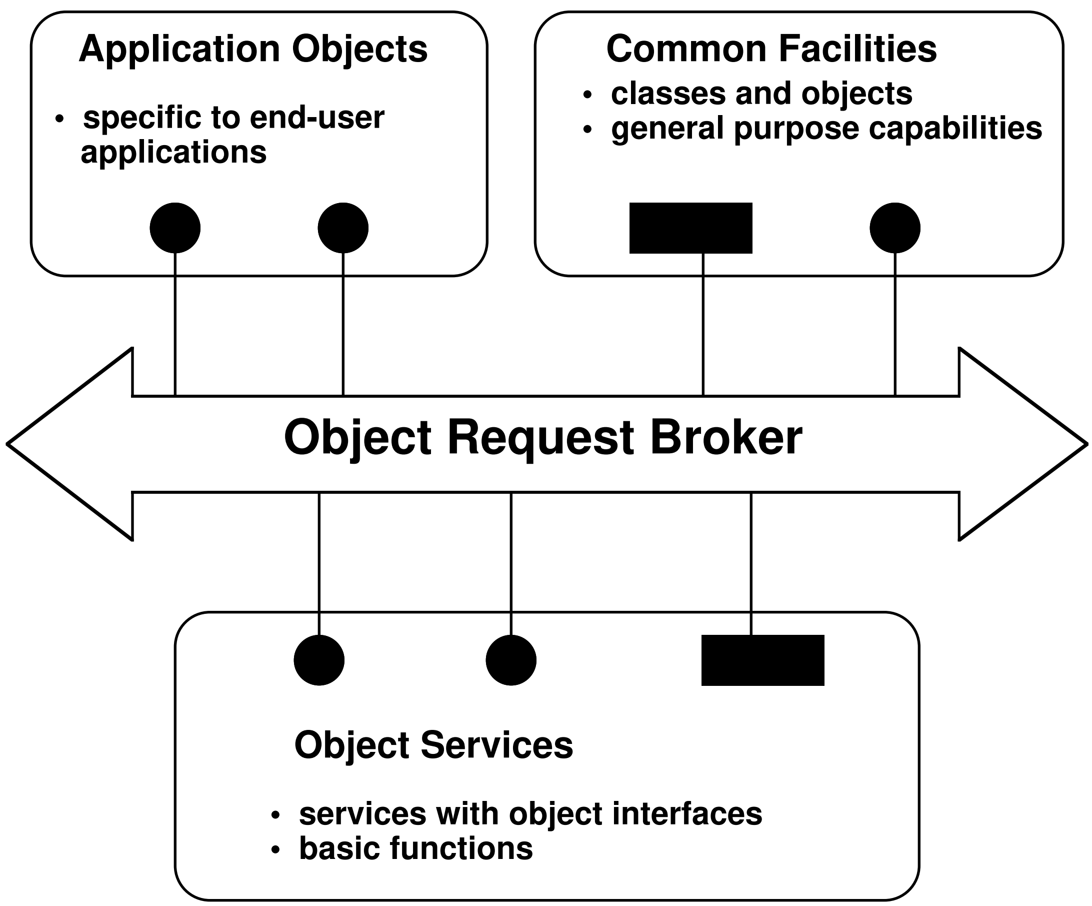

Figure 5.3.1. The OMG Object Management Architecture (OMA)

In the OMA, an *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a *client* (which can be another object, or a non-object-oriented application). An *operation* is an identifiable entity that denotes a service that can be requested. An operation has a *signature* that describes the legitimate values of request parameters and returned results. The signature consists of a specification of parameters, the result of the operation, a specification of the exceptions that may be raised by a request and the types of parameters accompanying them, a specification of any contextual information (e.g., client preferences that may be used to determine how to handle the request) and an indication of the execution semantics the client should expect.

The Object Request Broker (ORB) component of the OMA supports clients making requests to objects, as shown in Figure 5.3.2 [OMG91]. A *request* consists of an operation, a target object, zero or more parameters and an optional request context. A request causes a service to be performed on behalf of a client and any results of executing the request to be returned to the client. If an abnormal condition occurs during execution of the request an exception is returned. This is a form of the general concept of object communication via messages as described in Section 2.1. The ORB is responsible for finding the object implementation corresponding to the target object specified in the request, invoking that implementation, passing it the request for handling, and returning the results.

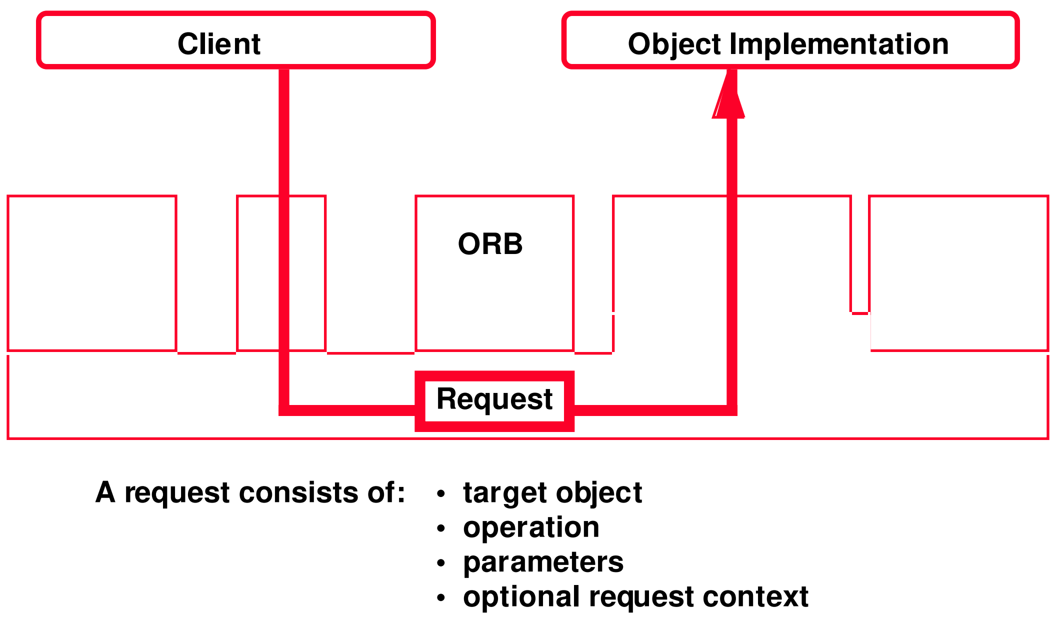

Figure 5.3.2. A Request in the OMG Architecture

5.3.2 OMG CORBA

The Object Request Broker (ORB) component of the OMA is defined by the OMG Common Object Request Broker Architecture (CORBA) [OMG91]. The structure of the CORBA architecture is shown in Figure 5.3.3 [OMG91]. The CORBA specification defines a number of different interfaces, as shown in the figure.

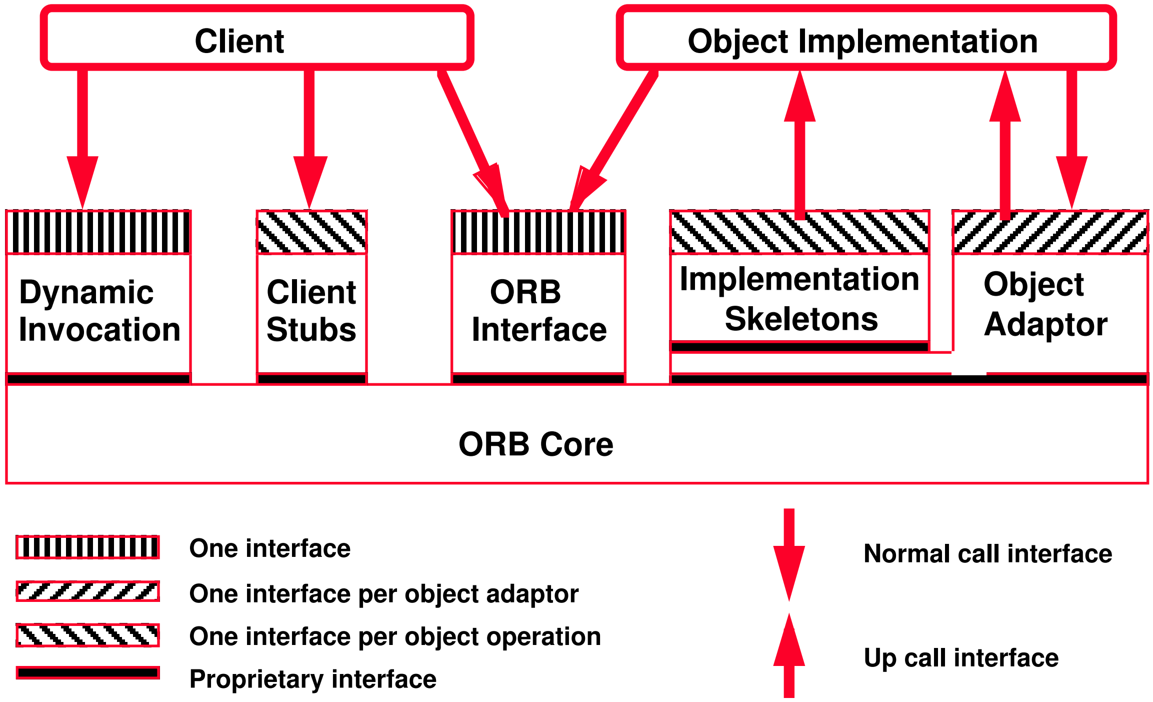

Figure 5.3.3. OMG's CORBA Architecture

The CORBA specification defines an Interface Definition Language (IDL) for defining the interfaces of objects in the architecture. IDL provides syntax for specifying the signatures of externally-visible operations that can be performed by an object supporting that interface. The CORBA IDL syntax is generally based on C++, and a host language binding to C is defined in [OMG91].

The CORBA IDL plays a role similar to that of the IDL in a Remote Procedure Call facility. IDL specifications can be compiled to produce *client stubs* (for clients to call object operations) and *implementation skeletons* (essentially server stubs); one stub and skeleton is defined for each operation supported by an object. The use of a compiled client stub to access an object interface is referred to as the *static* interface. A Dynamic Invocation Interface (DII) is also provided that allows clients to construct requests at run-time to objects whose types might not have been known at compile-time. Both static interfaces and the DII use an object's implementation skeleton on the object (server) side; static and dynamic requests are semantically equivalent methods of invoking operations on objects.

Objects are made known to the ORB by being registered with an *object adaptor*. The object adaptor uses the implementation skeletons of the various operations defined by the object implementation to call these operations when handling a request directed at the object. The definition of a Basic Object Adaptor (BOA) is contained in CORBA. The BOA is generally designed to handle objects that are independently constructed, and must be handled individually by the adaptor. The CORBA specification also identifies other types of adaptors that might be more appropriate for objects implemented in different ways. For example, CORBA identifies a DBMS object adaptor that would be more appropriate for objects defined within a database environment. (For example, in this case the object adaptor itself would not have to register individual objects as they were created, nor would it have to invoke an object method itself in response to a message directed at an object. Instead, created objects would be registered with the containing database system, and the object adaptor would merely forward any messages directed at database objects to the database system). As noted in Section 5.1, an Object Database Adaptor interface definition is part of the ODMG specifications.

The CORBA specification also defines the concepts of an *implementation repository* and an *interface repository*. The implementation repository is a database of information about object implementations that can be used by an object adaptor. The interface repository is a database of object interface definitions currently known to the architecture that can be used in defining new applications, or accessed at run-time by clients to construct dynamic requests.

5.3.3 CORBA IDL

As noted above, the CORBA specification defines an Interface Definition Language (IDL) for defining the interfaces of objects in the architecture. IDL permits interfaces to objects to be defined independently of an object's implementation (IDL provides no facilities for defining object implementations, leaving this to individual programming languages). As noted in Section 5.1, the ODMG ODL is a superset of CORBA IDL.

A CORBA IDL interface definition consists of a set of operation signatures, together with a (possibly empty) set of type declarations defining new types for use within the interface declaration. As noted earlier, an operation signature includes the specification of an operation name, a set of input parameters, a result, and the exceptions that can be raised by the operation.

Some example CORBA IDL interface definitions are:

```
Interface Stock {
   exception not_enough_money { float amt_needed; };

   void buy (in long Count)
      raises (not_enough_money);

   void sell (in long Count)
   readonly attribute float currentPrice;
}

Interface foo {
   enum material_t {rubber, glass};
   struct position_t {float x,y};

   attribute float radius;
   attribute material_t material;
   readonly attribute position_t position;
   ...
};
```

Types are used in operation signatures to restrict a possible parameter or to characterize a possible result. An object type is a type whose members are objects. Basic data types include `long`, `short`, `unsigned long`, `unsigned short`, `float`, `double`, `char`, `boolean`, `octet` (guaranteed to not undergo any conversion during transfer between systems), and `any` (a type any which can represent any possible basic or constructed type).

Constructed types consist of structures (type `struct`), consisting of an ordered set of (name, value) pairs, a discriminated `union` type consisting of a discriminator followed by an instance of a type appropriate to the discriminator value, and enumerated types (type `enum`) consisting of ordered sequences of identifiers. Template types include a `sequence` type which consists of a variable-length array of a single type, and a `string` type which consists of a variable-length array of characters. IDL also supports an array type, which consists of a multidimensional, fixed-length array of a single type, and an interface type (referencing the name of a defined interface) which specifies the set of operations which an instance of that type must support.

Interfaces defined in CORBA IDL can also be used as types. The identifier that names an interface defines a legal type name. A parameter or structure member defined as an interface type can hold a reference to an object supporting that interface.

A (pseudo-) interface `Object` is (effectively) predefined. This interface defines a set of operations on object references that are implemented by the ORB (rather than by the object implementation). The actual mapping of this interface depends on the language binding.

As the examples illustrate, an interface may have *attributes*. An attribute is logically equivalent to declaring a pair of accessor functions: one to retrieve (get) the value of the attribute and one to set the value of the attribute. An attribute may be defined as `readonly`, in which case only the get accessor function is defined.

CORBA IDL supports interface inheritance, meaning that one interface can be derived from another. A derived interface may declare new elements. In addition, the elements of a base interface (parent interface) can be referred to as if they were elements of the derived interface. A derived interface may redefine any of the type, constant, and exception names which have been inherited. An interface may be derived from any number of base interfaces (multiple inheritance). Reference to base interface elements must be unambiguous. It is illegal to inherit from two interfaces with the same operation or attribute names. CORBA IDL does not address object implementation inheritance.

CORBA does not define any specific objects. However, CORBA does define *pseudo objects*. Pseudo objects have IDL-defined interfaces just like ordinary CORBA objects, but may be implemented directly in the ORB instead of as ordinary CORBA objects. Pseudo objects defined as part of the CORBA specification include the ORB itself.

5.3.4 OMG Object Model Structuring

The OMG has recognized the problems of defining a common object model that is flexible enough to address the requirements of the many different kinds of objects that may participate in an open, distributed object architecture, and has defined a way of *structuring* their common object model into a "Core + Components" in order to address them. This approach (illustrated in Figure 5.3.4, a modification of one in [Cat94a]) can be described as follows:

1. The *Core* Object Model is a consensus of the least common denominator of atomic features that the OMG membership feels must be supported by any system that calls itself "object technology." Those features are identity, typing, operations, and subtyping/inheritance. The features are considered "atomic" in the sense that they cannot be further broken down. For

example, subtyping must be included in its entirety as defined in the Core in any OMG-compliant object model.

2. *Components* are compatible extensions to the Core that may be needed in some application domains, but not others. Examples of components are exception handling, attributes, and relationships. Components provide a way to define new extensions to the Core. For example, an exceptions component would extend the signatures of operations. Components can also be used to define new types to be added to the model without extending the Core Object Model semantics.

3. *Profiles* are collections, consisting of the Core plus one or more Components, that together make up a useful object model for a specific domain. These domains can be technology-specific (ODBMS, GUI, programming language, etc.) or application-specific (manufacturing, network management, finance, etc.).

4. Only Profiles make up useful object models, and compliance will be measured against a Profile. Compliance to just the Core or individual Components that do not make up a Profile is irrelevant.

As noted above, components must be compatible with the Core. They can extend the semantics of the Core but they cannot restrict the functionality defined in the Core. Also, components cannot be incompatible with each other. However, a component can be a refinement or extension of another. This appears to indicate that there can be alternative components providing the same facility (e.g., different versions of exception handling) *only* as long as one extends the other; they cannot be fundamentally incompatible. A number of possible components have been discussed within OMG, including the exception handling, attributes, and relationships components mentioned above, and also others such as non-objects (value types), meta-data, collection types, constraints, and extended operation semantics. Also, as noted in Section 5.1, the ODMG specifications include the definition of an ODBMS profile, including those components which, together with the OMG Core Object Model, make up the ODMG object model. A high-priority activity is the development of an object model profile covering the requirements of the CORBA IDL, since to a certain extent the development of CORBA IDL and the "Core + Components" approach proceeded in parallel. The OMG has also indicated that, at some later time, features contained in Components could be moved into the Core, if the features become so generally available in object models that the "Least Common Denominator" effectively changes.

The OMA explicitly states that the Core Object Model is not a metamodel (a model for describing or deriving other object models), and thus components defined as extensions to the Core do not have to be derived from Core concepts. In fact, the Core does not contain the necessary facilities to allow this. Moreover, the Core deliberately says nothing about some aspects of the model, in order to provide for maximum flexibility in adding Components. However, the "Core + Components" approach potentially provides a disciplined way of adapting the model to support specific application domains, while maintaining compatibility among added features.

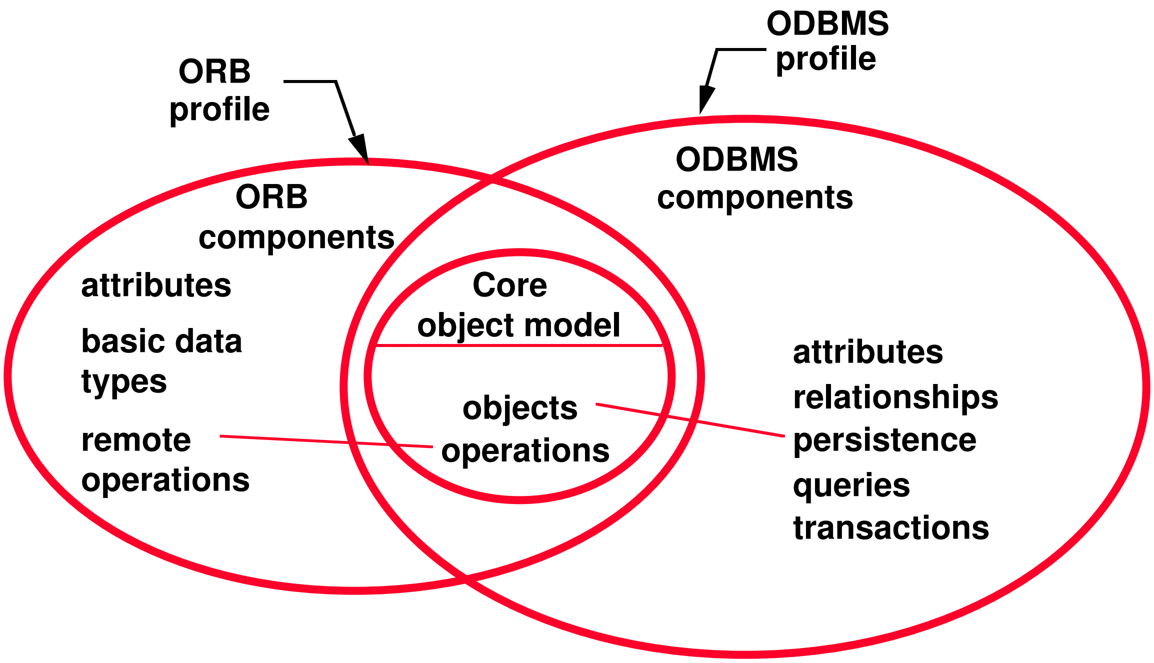

Figure 5.3.4. OMG's "Core + Components" Object Model Structure

5.3.5 OMG Object Services

As described in [OMG92b], an Object Service defines interfaces and sequencing semantics that are commonly used to build well-formed applications in a distributed object environment. Each Object Service provides its service to a set of users. These users are typically Application Objects or Common Facility objects that, in turn, provide support for specific application domains like word processing, CAD, or network management. However, Object Services can also be used by other Object Services.

In non-object software systems, a system's Application Programming Interface (API) often is defined by a monolithic interface. The OMG Object Services API is modular; particular objects may use a few or many Object Services. By being object-oriented, the OMG Object Services API is extensible, customizable, and subsettable; applications only need to use services they require.

A service is characterized by the interfaces that it provides, the objects that provide those interfaces, and, indirectly, by aspects of its implementation that impact its use (e.g., semantic assumptions, performance). A service may involve a single object (e.g., a time server object), multiple objects all of which provide the same interface type (e.g., thread objects), or multiple objects which provide distinct interface types that inherit from one service's interface type (e.g., all objects providing the lifecycle service). Finally, a service may involve combinations of these cases (e.g., a transaction service that defines transaction objects and also defines interfaces that are inherited by any "transactionable" object).

The operations provided by Object Services are made available through the CORBA IDL or through proposed extensions to IDL compatible with the OMG Object Model. However, while OMG requires an IDL interface for each object service, implementations of object services need not themselves be object-oriented. Such implementations may continue to support non-object-oriented interfaces for compatibility with an existing product's API or with non-OMG standards, in addition to an IDL interface. Also, objects need not use the implementation of basic operations provided by Object Services, nor must objects provide all basic operations. For example, an object may provide its own data storage; an object that models a process may not provide transactions.

Service	Function
Concurrency	Supports concurrent access to one or more objects by one or more objects
Event Notification	Supports asynchronous notification of events to interested objects
Externalization	Supports object externalization and internalization
Implementation Repository	Supports the management of object implementations
Interface Repository	Supports the management of object interface definitions
Lifecycle	Supports object create, delete, copy, and move
Naming	Supports mapping between names and objects
Persistence	Supports persistent storage of object state
Properties	Supports association of attribute values with an object that are not part of the object's interface
Query	Supports operations on sets and collections that return sets and collections. Supports indexing.
Relationships	Supports associations between two or more objects, together with referential integrity and cascaded operations.
Security	Supports access control on objects.
Transactions	Supports atomic execution of one or more operations
Version and Configuration Management	Supports identification and consistent evolution of object instances, structures, and their definitions

Table 5.3. Categories of OMA Object Services

Various lists of Object Services have been produced at various times by OMG. Table 5.3 summarizes the Object Services that are similar to those provided by an ODBMS, and that have been the subject of actual RFPs by OMG. Of these, the Event Notification, Lifecycle, and Naming Services are specified in [OMG94a]. A Persistence Service specification has been approved, and will appear in a forthcoming update of [OMG94a]. RFPs for the other services have been issued, and responses to several of those RFPs (containing candidate specifications for those services) are currently being evaluated by OMG.

Several of these services are of particular interest in the context of ODBMSs. Only brief descriptions are possible here. The OMG *Persistence Service*, which has already been approved by OMG, defines multiple interfaces and protocols designed to provide help in implementing various kinds of object persistence, and provide for varying degrees of client control over persistence. The general architecture of the Persistence Service is shown in Figure 5.3.5 [CP93].

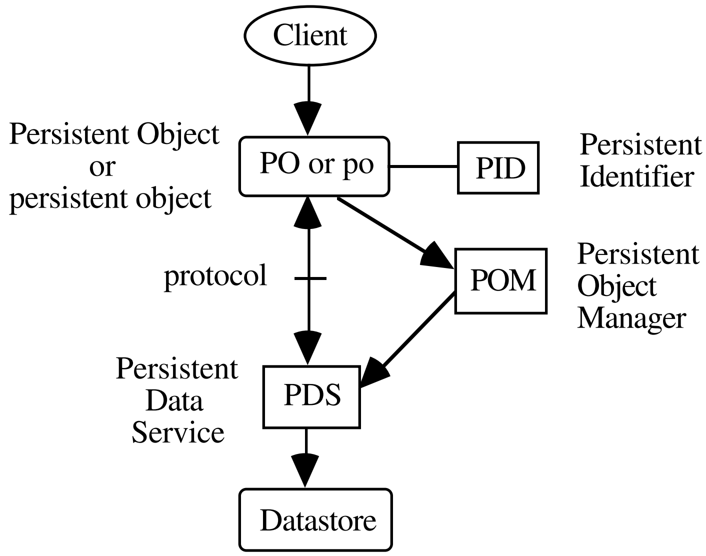

Figure 5.3.5 Persistence Service Architecture

In this architecture, a *Persistent Object* (PO) is defined as an object that exports its persistence behavior to clients. This behavior is provided via a Persistent Object (PO) interface that is separate from the object's ordinary functional interface. Through this interface, a client can control the synchronization of the object's state with separate units of persistent state stored in file systems or databases. An object may be persistent without explicitly exporting its persistence behavior to clients (i.e., without supporting the PO interface). In this case, it is referred to as a *persistent object* (po; note the lower case). Such objects would be used when clients do not need explicit control of the object's persistent behavior.

The Persistent Object interface includes the following components:

- an attribute `pid` of type `PID`; this holds the default PID for the object's persistent state.

- an operation `PDS connect (in PID pid);` this synchronizes the object's own state with its persistent state.

- an operation `void disconnect (in PID pid);` this synchronizes the persistent state with the object's own state and ends the connection.

- an operation `void store (in PID pid);` this synchronizes the persistent state with the object's own state.

- an operation `void restore (in PID pid);` this synchronizes the object's own state with its persistent state.

- an operation `void delete (in PID pid);` this deletes the persistent state.

Units of persistent state are identified by *Persistent Identifiers* (PIDs). These are objects that encapsulate the location of an object's persistent state, and provide the only way a client has of finding out where persistent state is stored, and the Datastore interface used to access it. The PID interface includes the following components:

- an attribute `datastore_type` of type `string`; this identifies a particular Datastore interface used to access the persistent object referred to by this PID (an example of a Datastore type might be Posix File System).

- an operation `string get_PIDString();` this returns a string form of the PID so that it can be externally stored.

- other attributes interpreted only by particular Datastores (e.g., file path and offset).

Using PIDs, the persistent state of an object can be referred to independently of the object itself. For example, the persistent state can be copied or backed up. Also, an object's PIDString can be obtained and given to another client, so that the client can share the persistent state without sharing the object itself.

The *Persistent Object Manager* (POM) defines a uniform interface for the implementation of an object's persistence. The POM implements the `connect`, `disconnect`, `store`, `restore`, and `delete` operations defined at the PO interface, except that these POM operations take the calling object's id as an additional parameter. A PO simply forwards these operations to the POM for handling. When used to implement a po, the POM performs the same functions, but the ability to explicitly call these operations is not exported to the object's client (instead, the object only uses the POM interface internally). The POM is responsible for routing persistence operations from a persistent object to the correct Persistent Data Service based on the protocol and the datastore type used by that object. A *Persistent Data Service* (PDS) supports one or more combinations of protocols and Datastore types. A PDS is responsible for implementing the `connect`, `disconnect`, `store`, `restore`, and `delete` operations supported by the POM, together with any additional operations defined by the specific protocols it supports.

Persistent Service *protocols* specify the way persistent data is moved into and out of an object. The actual details of the transfer mechanism depend on the protocol. Three protocols are defined. The *Direct Attribute protocol* allows an object implementor to define persistent data objects using a DDL (a subset of IDL supporting only attributes). Using the protocol, the actual CORBA object calls the PDS to get/put data between these data objects and itself. The *ODMG-93 protocol* allows an ODMG-93 database (as described in Section 5.1) to serve as a PDS. In this case, the data access protocol is defined by the ODMG-93 language bindings, and the CORBA object uses the facilities defined in ODMG-93 to interact with an underlying object database. In the *Dynamic Data Object (DDO) protocol*, the object inherits a special DDO interface which allows a PDS direct access to the object's persistent data. In this approach, the PDS calls the object to put/get data, instead of the object calling the PDS (as in the Direct Attribute protocol).

A *Datastore* provides an underlying storage mechanism for persistent state, e.g., a file system or database. Factory Objects (the mechanism defined in the Lifecycle Service for creating new objects) will provide the facilities for creating objects with or from particular Datastores. The Persistence Service specification defines one simple Datastore type, called DBDatastoreMgr. Other Datastore types are defined by other standards, including Posix files, ODBC for SQL, etc.

The OMG has also solicited responses to a Request for Proposal (RFP) covering an *Object Query Service* (OQS). This OQS RFP forms part of [OMG94b]. The OQS RFP does not impose a great number of detailed requirements on responses. It does, however, describe certain general characteristics of the proposed Query Service.

The RFP requires that object service interfaces be object-oriented, and expressed in IDL. Proposed extensions to IDL, CORBA, and the OMG object model are to be identified. The RFP indicates that it should be possible to specify queries in object derivatives of SQL, or in direct manipulation query languages (i.e., query facilities tightly integrated with a host programming languages, as in the ObjectStore query facilities described in Section 3.3.2). However, it notes that the SQL select-from-where construct may not be syntactically consistent with object-oriented method or operation syntax, even though it is very powerful.

The RFP also indicates that query operations should be extended with respect to traditional query languages in the following ways:

- Querying and/or returning complex data structures.
- Operating on user defined classes of objects.
- Operating on other kinds of collections as well as sets.
- Allowing the use of procedurally-specified operations in the query predicate.
- Making use of encapsulation, inheritance, or operation overloading
- Opening the query optimizer to allow new kinds of indices or new query algebras.

Three obvious candidate sources for OQS facilities are:

- the CORBA IDL, since object service interfaces must be specified in this language (or extensions to it).
- the ODMG-93 specifications for Object DBMSs (see Section 5.1), since these define an Object Query Language and IDL extensions for object databases.
- the SQL3 specifications (see Section 5.2), since these represent an object-oriented extension to the current SQL standard database language.

Section 5.4 compares these candidates (since they are major standards related to object DBMSs), and also evaluates their potential support for the OQS. The OMG expects responses to the OQS RFP to be submitted by October 1994. The OMG is also currently evaluating candidate submissions for a Transaction Service (and is near to selecting one).

5.3.6 Relationship of OMG Specifications to ODBMSs

It might not be immediately obvious why the OMG Object Management Architecture (OMA), and its associated specifications (CORBA, Object Services, etc.), are relevant to a discussion of ODBMS standards. There are two basic ways in which the OMA is relevant in this context. The first is as an extension of the way that applications interact with an ODBMS. Section 2.2 described the pattern of ODBMS access as shown in Figure 5.3.6.

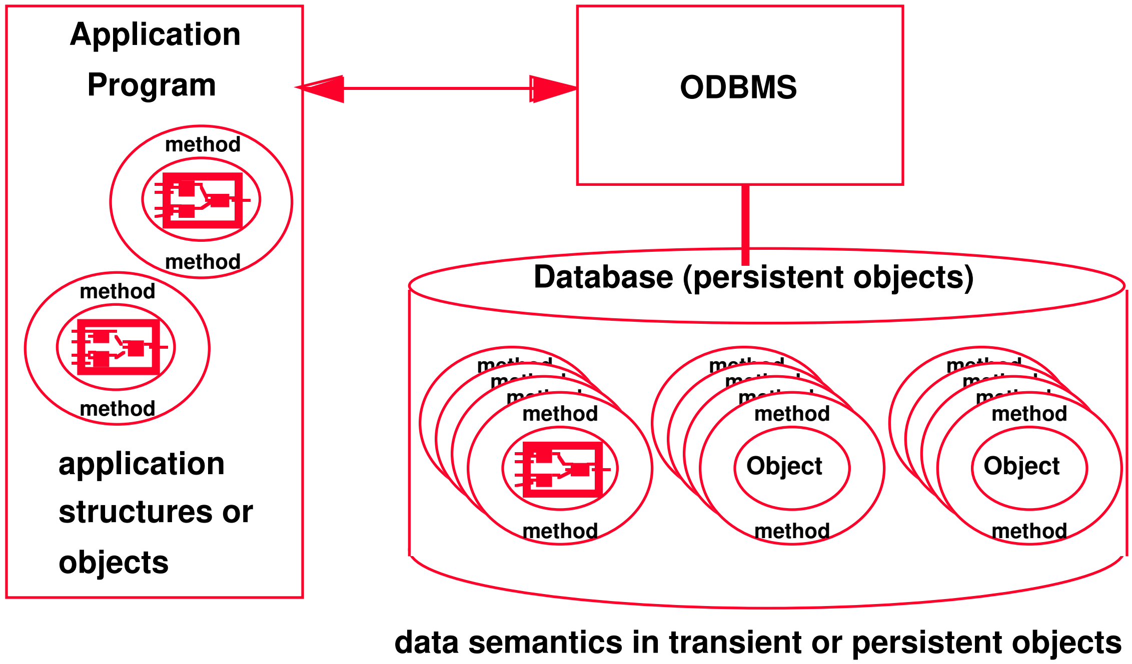

Figure 5.3.6. Application Access to an ODBMS

OMG's OMA and CORBA can be used to extend this approach by providing connectivity between applications and object DBMSs connected to the network, as shown in Figure 5.3.7. In this case, the Object Request Broker is used as an object-oriented messaging component external to the ODBMSs and client applications. Application objects have access not only to those ODBMSs to which they are directly connected, but also to any ODBMSs connected to the Object Request Broker. ODBMSs may also access other ODBMSs via this approach. As noted in Section 5.3.2, both the CORBA specification and the ODMG specification explicitly discuss a specialized DBMS adaptor specifically to connect DBMSs within CORBA.

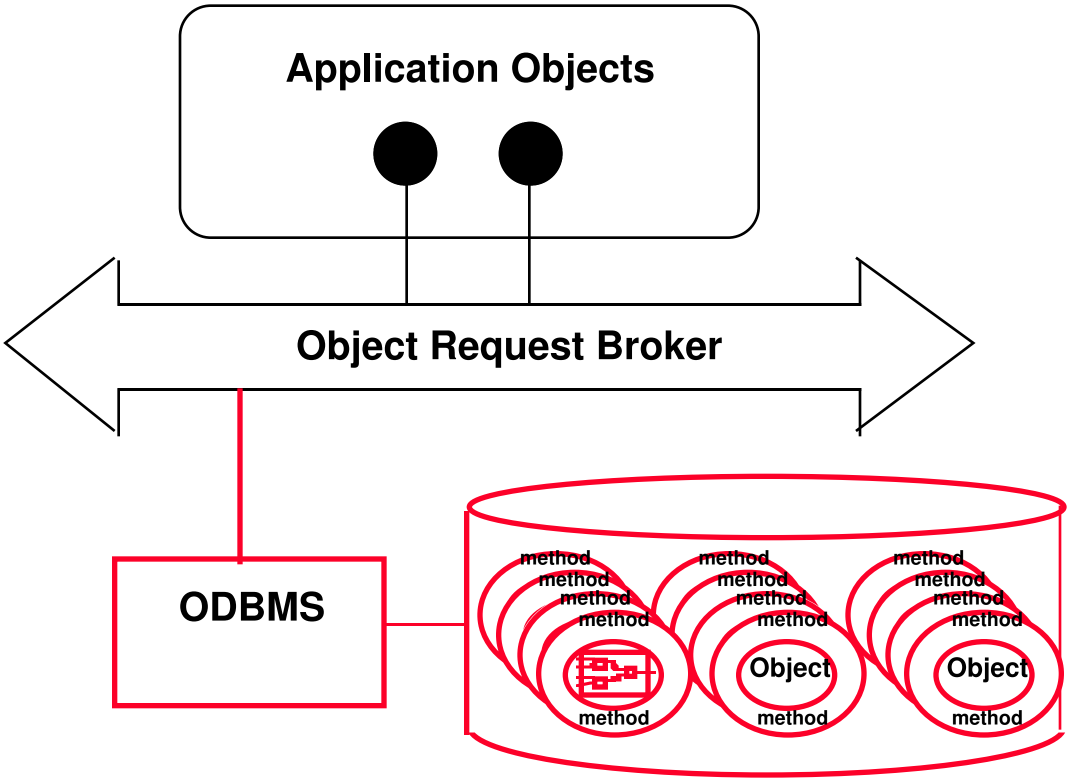

Figure 5.3.7. An ODBMS Attached in a CORBA Architecture

The second way in which the OMA is relevant in the context of ODBMSs is that the collection of objects within a distributed object architecture *can be considered as an object database itself*, provided appropriate services are available to provide ODBMS-like functionality, as shown in Figure 5.3.8. Managing such a collection of objects is effectively "object database management". The OMG-defined Object Services provide (or will provide) most, if not all, of the required facilities, such as transaction support, querying, persistence, etc.

In this view, CORBA becomes part of the internal implementation of what is effectively an ODBMS, serving to connect the various Object Services which together provide ODBMS functionality, rather than being a communications mechanism external to an ODBMS. The Open OODB prototype described in Section 3.9 is an example of the use of an OMA-like architecture to construct an ODBMS as a collection of object services connected by a messaging backplane. From this point of view, the specifications being developed by OMG for CORBA IDL, the various Object Services, etc., could be considered as candidate ODBMS standards. However, in some cases, these are standards specifying aspects of an ODBMS that are not seen by users, but instead are only of interest to implementors (e.g., the CORBA Object Adaptor interface). This is because the intent of OMG is to allow for open distributed object architectures integrating components implemented by *multiple* vendors, rather than to only specify the user-visible characteristics of an ODBMS that would be implemented by a single vendor.

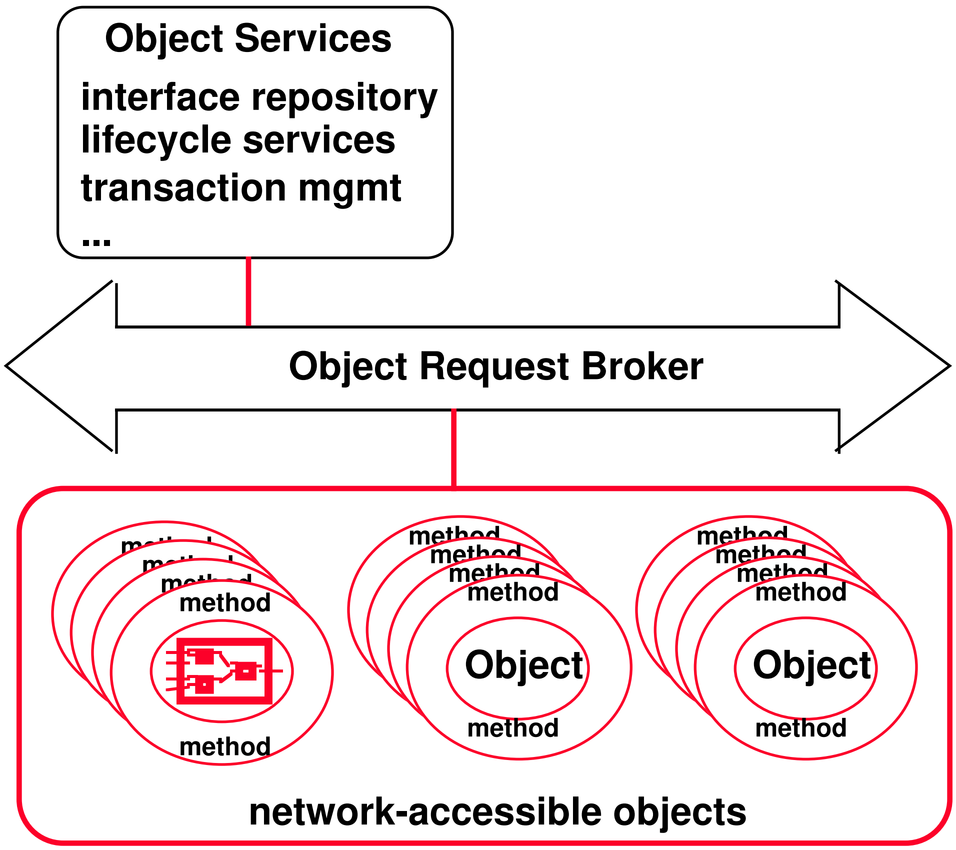

Figure 5.3.8. A CORBA Architecture Used to Create an ODBMS

5.3.7 Remarks

The OMG OMA, CORBA, and Object Service specifications provide the potential basis both for the development of ODBMS implementations, and also for the development of the distributed object architectures in which ODBMSs would likely be used. OMG specifications have already had great influence in defining the course of object standards, and numerous commercial products based on them have been produced. While these products are not necessarily fully mature, and much work remains to be done, the OMG specifications represent an approach around which many software vendors are rallying.

The next section discusses the ODBMS standards that have already been presented, compares and contrasts them, and discusses the possibility for a merger of, or interoperability between, these standards.

5.4 Standards Discussion

The preceding sections have described the ODMG-93, SQL3, and OMG specifications as being key standards relevant to ODBMSs. This section compares these standards, and discusses possibilities for merging the standards, and for ODBMSs to support interoperation among them. The comparison primarily concentrates on the *object models* supported by the standards, since the object model forms the basis for the other facilities, such as query facilities, supported. Hence, for example, much of the discussion of the

OMG standards centers around OMG's IDL. However, other aspects of the standards are also considered. Much of this material is extracted from [MM94], a paper submitted to the X3H7 Object Information Management standards committee, and which is also being submitted to ODMG, OMG, and X3H2 (the SQL standards committee), with the idea of trying to address differences among these standards.

5.4.1 Comparing the Standards

The following subsections compare and contrast the ODMG, OMG, and SQL3 standards, using for the most part object model features defined in [Man94] as the basis of comparison. In some cases, the comparisons involve only ODMG and SQL3. This is so because the ODMG object model is designed as a strict superset of the IDL object model, with additional facilities intended for use by Object DBMSs. Where there is a significant difference between ODMG and IDL, this is noted.

5.4.1.1 Objects

Both the ODMG and SQL3 type systems allow both mutable and immutable user-defined types, and also structured types of objects and literals that parallel each other to a certain extent. CORBA IDL is more limited, in that it allows only user-defined mutable types, and is also limited in the structures that are supported (these limitations could be addressed by Object Services). It seems as if the parallels between ODMG and SQL3 could easily be the basis for defining a "relational subset" of OQL that could, if necessary, have specially optimized syntax for this subset that would be similar, if not identical, to SQL.

ODMG appears to more fully integrate objects and literals; for the most part, arbitrary mix-and-match of objects and literals is possible. In SQL3, the integration of ADT and table facilities is not as tight, and there currently appear to be some limitations on the flexibility with which the concepts can be intermixed. For example, ADTs must be in tables to be persistent, and multisets of row types are close to, but not identical to, tables. At the same time, it is not clear how much ODMG has really thought about user-defined atomic literal types (aside from saying they can be defined).

As currently defined, SQL3 effectively requires a table at the top level of a database schema; i.e., tables (and routines) are the things that have independent existence in the database. ODMG is less clear about this. On the one hand, the ODMG object model indicates that objects can have multiple names, but is silent about whether literals can have names. On the other hand, the OQL definition indicates that literals can have names. It appears possible to define ODMG structures that fully subsume the structures that can be defined in SQL3. If objects are the only thing that have names in ODMG, an object would have to be defined to represent each SQL3 table. The extent of the table could then be defined as a multiset (or other collection) of tuples. If such an extent could also have a name (as suggested by the OQL specification), the mapping is even more direct. The models would be more parallel if things besides tables could have names and be persistent in SQL3, and if the table-oriented and ADT-oriented facilities in SQL3 were more thoroughly integrated into the same type system (recent work on SQL3 illustrates progress in this direction).

5.4.1.1.1 operations

ODMG ODL and CORBA IDL are compatible in their specifications for operations. Operations are defined on individual types. The interface of a type includes operation signatures: argument names and types (arguments may be in, out, or inout), possible exceptions, and result types. The first argument of an operation is distinguished (i.e., dispatching considers only this argument).

Operation names may be overloaded; dispatching is based on the most specific type of the first argument of the operation call. Operations may have side-effects; an operation with only side-effects can return a nil value. The ODMG model assumes operations will be executed sequentially; it does not require support for concurrent or parallel operations.

ADT operations in the SQL3 model are similar in many respects. Differences are that possible exceptions are not part of operation signatures, and operations do not have distinguished arguments; dispatching in SQL3 is based on all argument types, using a "most specific routine" concept. Operations may have side-effects. SQL3 also supports operations that are independent of ADTs or other specific data structures. Such operations can be defined to take arguments from tables with row identifiers defined, in which case the operations somewhat resemble operations on objects, and the operations in a subtable hierarchy can be overloaded.

In ODMG, implications for optimization are that optimizers must be conservative since an operation in a query might have side-effects. They suggest a pragma distinguishing operations that can safely occur in query expressions. This is an idea that SQL3 might also consider.

5.4.1.1.2 methods

In ODMG and IDL, operations (declared in the interface) are implemented by methods defined in the type implementations. Each operation is implemented by a method; an implementation might also define other methods. ODMG's ODL, like CORBA IDL, defines only interfaces, hence all object characteristics are, in SQL3 terms, PUBLIC. Mechanisms for specifying methods are not explicitly provided in ODL; however, methods could be specified using the ODMG language bindings. SQL3 also defines ADT implementations as well as interfaces, and hence includes other encapsulation levels, and mechanisms for actually defining (or referring to) aspects of the implementation. ADT operations are implemented by routines, which may be defined either entirely in SQL, or in arbitrary programming languages. SQL3 also supports operations that are independent of ADTs or other specific data structures. Such operations can be defined to take arguments from tables with row identifiers defined, in which case the operations somewhat resemble operations on objects. SQL3 includes extensions to make it computationally complete; ODMG has not taken this approach.

5.4.1.1.3 state

In ODMG, object state is modeled by the properties of an object. A property can be an attribute or a relationship. The attributes and relationships of an object are defined as part of the type interface. Attributes take literals as their values; relationships can only be defined between two nonliteral object types. State in both ODMG and IDL is *abstract* state; there is no implication that attributes or properties are necessarily implemented directly as stored data. ODMG's ODL, like CORBA IDL, only defines interfaces, hence all object characteristics are, in SQL3 terms, PUBLIC. Mechanisms for specifying object implementations (including internal state) are not explicitly provided in ODL; however, object implementations (including internal state) could be specified using the ODMG language bindings. SQL3 defines both the interface and the implementation of ADTs, and hence includes multiple encapsulation levels. SQL3 allows the definition of how ADT attributes will actually be implemented. For example, defining the attribute as VIRTUAL means that the state defined at the object interface is actually abstract; otherwise the attribute will be represented as stored state. Both ODMG and SQL3 also support non-object state, in the form of literals associated with schema-defined names (e.g., rows in tables in SQL3). Relationships are modeled in SQL3 by ADT attributes containing identifiers of ADTs or rows, or by tables.

5.4.1.1.4 object lifetime

In ODMG, object lifetime is orthogonal to type, is specified at object creation and, once specified, cannot be changed (although there are currently some minor inconsistencies in the specifications relating to lifetime). Lifetime can be "coterminus with procedure" meaning it is declared in the heading of a procedure, allocated out of the stack and returned to the free pool when the procedure ends, "coterminus with process" meaning it is allocated by the programming language runtime, or "coterminus with database" meaning it is managed by the DBMS runtime. Lifetime is not applicable to literals (immutable objects). Literals always exist implicitly. Most queries return literals. The OQL query language contains expressions for constructing objects, but there is no mention of lifetime of these objects. ODL allows names to be assigned to individual persistent objects.

IDL does not support object creation using an operation applied to a type, but rather uses a "factory object" approach defined by the Lifecycle Service. The multiple lifetimes specified by ODMG could presumably be implemented using OMG Object Services by one or more factory objects offering these options. Persistence could similarly be provided using the Persistence Service (which, in turn, could rely on ODMG facilities).

SQL3 currently requires that ADT instances, to be persistent, must be stored in tables. These instances can be of any defined type. There is no facility for assigning a name to an individual persistent object, except via the table mechanism. A row in a table exists until it is deleted (rows with identifiers are in some respects similar to objects, and in some respects similar to literals).

5.4.1.1.5 behavior/state grouping

Both IDL and ODMG define *classical* object models; each operation has a distinguished argument. SQL3 defines a *generalized* object model; operations are dispatched using all the arguments of an invocation to determine the appropriate routine to execute. Currently, a routine defined within an SQL3 ADT has privileged access to the implementation of any of the other argument ADTs. A generalized model may be necessary for SQL3 in order for routines to operate on tables correctly; however, this needs to be examined more carefully, since it is not clear how easy it will be for programmers of SQL3 databases to actually follow the dispatching rules, and ensure that the appropriate routine can always be identified distinctly. Discussion in OMG during the early development of the OMG object model raised a number of issues concerning the use of a generalized model in a distributed environment which would need to be examined again if the models were to be merged.

5.4.1.2 Polymorphism

Both the ODMG and IDL models support the polymorphism implicit in subtyping and operator overloading. ODMG supports parameterized collection types as well. SQL3 supports similar polymorphism for both ADTs and for tables in subtable hierarchies. In addition, SQL3 also supports *template types*, allowing the specification of user-defined parameterized types (which may be atomic). There appears to be nothing corresponding to user-defined template types in the ODMG specifications (only the predefined parameterized types).

5.4.1.3 Encapsulation

In ODMG and IDL, objects are instances of a type which specifies the interface for accessing the object. There is only one interface for a type. SQL3 supports the PUBLIC, PRIVATE, and PROTECTED encapsulation levels for ADTs, effectively creating different interfaces for different "clients" (users of the type, the implementation of the type, and the implementation of subtypes of the type). SQL3 supports encapsulation for tables to the extent that views are considered as providing encapsulation.

5.4.1.4 Identity, Equality, Copy

In ODMG (and in IDL), all denotable objects have an identity. For literals this identity is "typically the bit pattern that encodes its value". For objects, identity "uniquely distinguishes the object from all other objects within the domain in which the object was created" and is independent of any state of the object. In ODMG, the operation `Equal?` is defined for type `Denotable_Object`. IDL does not guarantee that comparing object references is necessarily equivalent to a test for identity. ODMG allows overriding the definition of equality and ordering, although no syntax is explicitly provided to do this.

In SQL3, OBJECT ADTs have OIDs, and are similar to ODMG objects in that respect; VALUE ADTs are similar to ODMG literals in having a state-based identity (although in both types of ADTs, the equality operation can be redefined by the user). Rows in tables having row identifiers have row identifier values, and are also somewhat similar to objects. In SQL3, for OBJECT ADTs, it is also possible to specify whether an ADT reference refers to an actual ADT instance (which includes its OID), or the OID of an ADT instance located elsewhere. Specifically, whenever an OBJECT ADT is used in a type definition, e.g., in a column definition of a table, or in a variable definition of a procedure or function, the ADT type reference consists of the ADT name, followed by an optional *elaboration mode*, represented by the keyword INSTANCE. If the ADT is an OBJECT ADT and the reference does not specify INSTANCE, then the data item (column, variable value, etc.) contains an OID that identifies an instance of the abstract data type. Otherwise, the item contains an actual instance of the ADT. This allows for both the usual *reference* semantics found in many object models, as well as for the specification that an object (instance) is to be *embedded* within another ADT instance, or to physically reside in a column of a table. SQL3 also allows OBJECT ADTs to control whether OIDs are VISIBLE or NOT VISIBLE.

This is a place where some additional work might be done to integrate objects into SQL in a cleaner way (for example, if objects could be persistently stored independently of tables, then table columns could always use reference semantics, and the distinction above could be eliminated). Work in this area is currently in progress. For example, a proposal [Sha94] has been submitted that associates OIDs with the "site" of an ADT instance (i.e., the variable, column, attribute, or parameter) as opposed to associating the OID with an attribute of an ADT instance. Another proposal [Kel94c] has suggested removing OBJECT ADT's entirely, instead using a combination of VALUE ADTs and row identifiers to provide the same facilities.

The ODMG specification of a type also allows the specification of keys, which are properties or sets of properties the values of which uniquely identify instances of the type. SQL3 allows specification of such primary keys for tables.

ODMG also allows names to be associated with objects. In the description of the ODMG object model, the names associated with an object are accessible via a predefined attribute `names` defined for type `Object`. However, the OQL description indicates that names can also denote any (denotable) object, including literals, although how this is done is not clear. SQL3 only allows names to be associated with tables and routines.

5.4.1.5 Types and Classes

In IDL and ODMG, a type is a specification; it can have one or more implementations. All ODMG types are instances of type `Type`. A class is the combination of a type specification and a specific implementation. The models are strongly typed. Two objects are compatible if they are instances of the same declared type or if one is an instance of a subtype of the other. In ODMG, two structured literals have the same type if they have the same structure at every level and the corresponding atomic types are the same. Subtyping for structured literals requires the same structure at each level and the type of each subobject of the subtype to be the same as, or a supertype of, the corresponding subobject of the supertype. No implicit conversions are given for either objects or structured literals. Some explicit conversion expressions are defined in the ODMG OQL specification. ODMG, unlike IDL, allows a type interface to specify that an extent is to be maintained for instances of the type. The extent may also have defined keys.

In SQL3, an ADT is both a specification and an implementation. DISTINCT types with the same implementation can be specified, but it is not clear how one would specify multiple implementations for the same type. SQL3 allows the definition of CAST functions for converting between types. In addition, a number of built-in type conversions are defined. ADT values cannot be persistent outside of tables. The extent of an ADT would have to be explicitly created (and populated) by the user. SQL3 also supports the table construct, which involves the concept of a row type (the sequence of column-name/type pairs), as well as the explicit definition of row types as one form of data type. Subtyping for row types is under consideration, and a form of subtyping for rows in the tables of a table hierarchy (based on the subtyping defined for the row identifier types of these tables) is supported.

5.4.1.6 Inheritance

The IDL and ODMG models define only type inheritance (subtyping); they do not define implementation inheritance. If S is a subtype of T, then S inherits all operations and properties of T, and S may define new operations and properties applicable to its instances. In other words, objects of type S have all the characteristics (and more) of type T. A subtype can specialize the properties and operations it inherits, but there are no rules given to indicate what kinds of refinement are correct.

In ODMG, a type can inherit from multiple supertypes, but must rename same-named inherited operations or properties. New subtypes may only be defined under type `Object`. Currently, no subtyping is allowed for characteristic types (attributes, relationships, operations). Type compatibility for objects is by name. Two objects have the same type if and only if they are instances of the same named type. An object of a given type can be assigned to an object of any of its supertypes. For literals, type compatibility is by structure - structured literals (immutable collections or structures) have the same type only if they have the same structure at each level and corresponding atomic parts have the same type. "Subtyping" of structured literals requires that both have the same structure at every level, and that the type of each 'subobject' of the supertype is the same or a subtype of the corresponding subobject of the subtype (thus, a subtype may not have *more* structure than its supertype). It is not clear whether or not subtyping of parameterized collection types is defined.

There are a few minor problems in the current ODMG definitions related to subtyping. For example, in one place, the specification says that type extents (the collection of all objects of a type) are predicate-defined collections where the predicate is `Is of type <T>`, where <T> is the type whose instances are the members of the extent. In another place, it says that "if type A is a subtype of Type B, then the extent of A will be a subset of the extent of B." It is logical to think that assignment to an extent would be made by checking the object's `type` property (one of the predefined properties defined for type `Object`). However, the specification also says that the `type` property returns only one type. Thus, asking an `Employee` instance for its type would return `Employee`, not `Person`, and hence this object would not be inserted into the `Person` extent according to that extent's predicate. This could be fixed relatively easily, e.g., by using a separate `Is_compatible_with <T>` predicate for the extent.

Subtyping for ADTs in SQL3 is similar to the description of ODMG subtyping above: two objects have the same type if and only if they are instances of the same named type. An object of a given type can be assigned to an object of any of its supertypes. A type can inherit from multiple supertypes, but must rename same-named inherited operations or properties. It is not clear whether or not subtyping of parameterized collection types (set, list, multiset) is defined. ADT subtyping in SQL3 also defines implementation inheritance (type and implementation inheritance are bundled together). SQL3 also supports table inheritance. The row type of a subtable is an extension of the row type of its supertables (and the row identifier type of a subtable is a subtype of the row identifier type of its supertables). Rules are also defined to keep the row membership in sub- and supertables consistent, so that they reflect the "containment" semantics expected of the extents of sub- and supertypes.

5.4.1.7 Noteworthy Objects

5.4.1.7.1 relationships

IDL does not define relationships as a separate concept (this would be defined by an object service). Instead, relationships are represented by attributes containing references to other objects. In ODMG, relationships are a kind of property defined between two mutable object types. Relationships are not objects. A relationship can be one-to-one, one-to-many or many-to-many, but are only binary. Relationships are defined in the interface(s) of the object type(s) involved in the relationship as a "traversal path" from one type to another. A two-way relationship will have a name in each type interface, and an `inverse` clause in each path declaration. Relationships define referential integrity and cardinality constraints on the participants in the relationship. Such constraints do not appear to be enforceable for structured values, as opposed to objects.

In SQL3, some relationships can be defined (in one direction) by ADT attributes having ADT OIDs as values. Table columns containing the row identifier values of other rows can similarly be used to represent relationships. Relations (tables) can be used to define generalized n-ary relationships; referential and other integrity constraints can be defined on these tables.

5.4.1.7.2 attributes

In ODMG, attributes are a kind of property defined on a single object type. Attributes take literals as their values. Attributes are accessed by get_value and set_value operations. They are defined as part of the type *interface*; there is no implication that they are represented by stored state in the implementation. Attributes in ODMG are not first class objects (they cannot have properties or be subtyped), however the built-in get_ and set_value operations can be overridden. Note that because attributes take literals as their values, defining, for example, a `works_for` "attribute" of a `Person` object having a `Department` object as its value requires the use of an ODMG *relationship* instead. (However, the OQL syntax for accessing the value of an attribute and traversing a relationship is the same.)

IDL attributes are similar to ODMG attributes, but IDL allows object interfaces to be used as attribute types (thus, in IDL one could have a `works_for` attribute of a `Person` type in the example above).

ODL allows the specification of attributes of types struct, enumeration, set, list, bag, and array in addition to the simple types (floating point, integer, character, boolean, etc.), sequence type, and string type that can be specified in IDL.

In SQL3, attributes of an ADT (or columns of a table) can be defined as having any predefined type, VALUE or OBJECT ADT type, row type, collection type, or type generated from a template.

5.4.1.7.3 literals

In ODMG, literals are immutable objects - either atomic (integer, float, Boolean, character) or structured. Structured_literals have two subtypes - Immutable_Collection (bit strings, character strings, and enumerations are built-in) and Immutable_Structure (date, time, interval). Additional subtypes can be defined, but operations on the built-in literal types cannot be redefined. The literal types are expected to directly map to types in the programming language. If the language does not have an analog for one of the literal types, the type must be defined in a library provided as part of the language binding.

ODMG supports many of the built-in types of SQL, including DATE, TIME, TIMESTAMP, and INTERVAL (and cites SQL as their definition).

In addition to its set of built-in types, SQL3 supports the definition of user-defined literal types as VALUE ADTs. It also supports literal collection types Set(<T>), Multiset(<T>), and List(<T>), and row types.

5.4.1.7.4 containment

Neither IDL nor ODMG explicitly support the concept of an object being contained within another (literals can be contained within literals or within objects). SQL3 supports the concept of an OBJECT ADT being contained within a table (and being referenced from other tables) through the use of the INSTANCE elaboration mode. SQL3 also supports the concept of literals being contained within literals (e.g., instances of row types, or collections of such instances, can be contained in a column of a row) or within objects.

5.4.1.7.5 aggregates

In ODMG, structured objects (aggregates) can be of type Structure or type Collection. Structures are records with named slots which can be filled by objects or literals of different types. Collections are homogeneous (modulo subtyping) groupings of elements that may or may not be ordered. Built-in collection types are set, bag, list, and array. Sets and bags are unordered collections; lists and arrays are ordered. Arrays are variable length (although an initial size is specified at creation) and can contain nil values. Collection types are instances of parameterized types. They can be parameterized by any subtype of Denotable_Object. There are two subtypes of collections: predicate_defined and insertion_defined. A type extent is a predicate_defined collection. Both mutable (object) and immutable (literal) collection types are defined. Mutable collections (subtype of Structured_Object) have intentional semantics. Immutable collections (subtype of Structured_Literal) have extensional semantics.

In ODMG, iterators can be defined to traverse collections. Type Collection also defines predicate-based select operations. Query operations apply to any collection (extents, user-defined). The result of a selection is a subcollection of the same type as the collection queried. Each of the more specific collection types defines appropriate query operations. Of course, the literal collection types do not define update operations (insert, delete, replace).

ODL uses the keywords Array and Sequence interchangeably to specify variable length, 1-dimensional arrays. (In IDL, Sequence is a keyword used for 1-dimensional arrays.) As in IDL, multidimensional array specification is implied by bracketed dimensions.

SQL3 provides row types as literal structures. Instances of row types can be used as values in tables; row types can also be nested. In SQL3, built-in collection types are set, multiset (bag), and list. Sets and bags are unordered collections; lists are ordered. Collection types are instances of parameterized types. They can be parameterized by any type. Tables are multisets of rows (literal structures), but are not considered as instances of these parameterized types (although the row type facility comes close to providing this integration, tables do not appear to be strictly defined as multisets of an explicitly defined row type). Set and List tables can also be defined.

An SQL3 collection can be used as a simple table in queries. In this case, each element of the collection corresponds to a row in the table. The table is treated as having a single column whose type is defined by the type of the instances of the collection. A collection is a VALUE (has no OID). Since collection types are data types, they must be declared as the types of table columns in order to store instances of collections persistently in the database.

5.4.1.7.6 other

In ODMG, type `Exception` is provided by the object model, and may be subtyped. Operation signatures can indicate exceptions that can be raised. When an exception is raised it is handled by the exception handler in the closest scope and transactions within that scope are aborted.

All three models support some variant of a "null". IDL provides an `OBJECT_NIL` object reference, which indicates that a reference identifies no object. There is no corresponding "null" defined for values. ODMG supports a distinguished `nil` denotable object that is used as both a literal and an object (e.g., it is assigned to an attribute to clear its value, and is also returned from an operation that has only side-effects). SQL3 supports a "null" for all types. The semantics of this "null" are built into certain query operations (e.g., comparisons, outer-join operations).

5.4.1.8 Extensibility/Introspection

In ODMG, type `Type` is a subtype and an instance of type `Atomic_object`. The metadata can be accessed using the interface for type instances, and can be queried using the OQL query language. Other metadata is discussed as extensions. ODMG provides no explicit facilities for schema evolution (however, a number of ODBMS products provide such facilities).

In SQL3, there are predefined tables containing the metadata associated with a schema, which can be queried as with any other table. SQL3 provides for a limited amount of schema evolution through an ALTER statement that can be used, for example, to add, alter, or drop columns from tables, and add or drop supertables and data types.

The CORBA specification [OMG91] defines an Interface Repository Interface that provides access to metadata about IDL defined interfaces and other information within the CORBA architecture.

5.4.1.9 Object Languages

5.4.1.9.1 Language Bindings

IDL has a C binding defined; C++ and Smalltalk bindings are under discussion. ODMG has bindings for C++ and (the beginnings of) one for Smalltalk, but none for other languages. The ODMG bindings define the "tight" sort of programming language interface found in many of the OODBMSs described in Section 3, i.e., a direct mapping of ODMG object model concepts to programming language object model concepts, support for both persistent and transient instances of programming language types, more-or-less transparent movement of objects between the database and application program, etc.

The SQL92 standard defines language bindings for a number of standard languages. These bindings assume a "looser" type of PL/DBMS interface (the assumption being that the type systems are in two different "spaces"). A key aspect of the individual language bindings is the definitions of correspondences between SQL data types and host language data types. In some cases, these are relatively straightforward; e.g., the SQL CHARACTER data type maps to a C char. In other cases, the mapping is not so straightforward. For example, SQL92 has a TIMESTAMP data type, but standard programming languages do not contain a corresponding built-in type. In these cases, SQL requires the use of a CAST function to convert database TIMESTAMP data to character data in the program, and vice-versa [MS93]. In SQL92, these type correspondences are defined only at the level of elementary scalar data types. There are no type correspondences defined for structured types, e.g., between a row of an SQL table and a flat record or structure in a programming language (although some such correspondences would be relatively straightforward to define).

There are currently no bindings defined between the SQL3 ADT extensions (or rows with row identifier values) and object classes or types in object-oriented programming languages such as C++ or Smalltalk. [BN94] is an X3H2 working paper that contains some preliminary discussion of such bindings for the ADT extensions. It notes that defining bindings between user-defined SQL ADTs and host language object classes would involve some form of *registration* of the mapping with the DBMS. [BN94] also notes the need to map between SQL OIDs and host language pointer or OID types.

[BN94] discusses both C++ and Smalltalk mappings. The mapping of the SQL ADT facility to C++ is more straightforward than the Smalltalk mapping, since a number of the concepts in the SQL ADT facility are based on C++ concepts. For example, like C++, SQL ADTs have multiple encapsulation levels (PUBLIC, PROTECTED, and PRIVATE). Similarly, both type systems support a distinction between a type that contains an instance of an ADT, and a type that contains a reference or pointer to the ADT. A Smalltalk mapping would be somewhat more complicated. For example, Smalltalk supports only public functions and private attributes (instance variables). Smalltalk only supports references to objects (OIDS); there is no concept of an embedded object (i.e., using the INSTANCE elaboration mode) [BN94].

5.4.1.9.2 Model-Defined Languages

OMG's IDL language is intended for defining object interfaces. As such, it provides syntax for specifying the types of externally-visible attributes and the signatures of externally-visible operations. IDL provides neither programming nor query facilities, the intent being that other languages will be defined and used to provide these facilities.

ODMG specifies an object definition language (ODL) that supports the ODMG object model and is compatible with OMG's IDL. The ODL is programming language independent. ODMG also specifies an SQL-like object query language (OQL) that provides declarative access to objects. Queries can be posed to any denotable object, starting with an object name or with a language expression yielding the object. For example, the query [Cat94a]:

```
select distinct x.age
from x in Persons
where x.name = "Pat"
```

searches over the set named `Persons` (in this case, the extent of type `Person`).

The nested query in:

```
select distinct struct(name: x.name, hps: (select y
      from y in x.subordinates
      where y.salary > 100000)
from x in Employees
```

searches over the result (a set of objects) of the query (path) expression `x.subordinates`.

The full OQL syntax is not currently supported by either the C++ or the Smalltalk binding.

SQL3 also provides both data definition language and query language facilities. In addition, new facilities in SQL3 are intended to support full programming capabilities.

5.4.1.9.3 Specific Requirements of the OMG Object Query Service RFP

As noted in Section 5.3, the OMG Object Query Service RFP (part of [OMG94b]) does not impose a great number of detailed requirements on responses. It does, however, describe certain general characteristics of the proposed Query Service. This section briefly reviews the candidate object models and languages with respect to those characteristics (since SQL3 is fundamentally a query language, query characteristics will play an important role in integrating the various standards).

1. Object-oriented OQL interface, expressed in IDL.

ODMG defines a mapping to IDL, plus IDL extensions. Some such mapping would have to be defined for SQL3.

2. Possibility of specifying queries in object derivatives of SQL, or in direct manipulation query languages.

ODMG provides a binding to an SQL-like syntax, even though the semantics differ in some cases. It seems as if it should be possible to define an SQL binding to the OQL semantics, since multiple bindings are possible (together with a corresponding binding of SQL data structures to a subset of the ODMG data structures).

SQL3 is, of course, an "object derivative of SQL". Further work should be done to merge the capabilities of SQL3 and ODMG in this respect. SQL3 does not consider a direct manipulation query language, using SQL embedding instead.

3. Querying and/or returning complex data structures.

ODMG provides a number of facilities addressing this requirement. The specifications are not clear in some of the details related to this requirement. It does not appear to be possible to return an entirely new object type (constructed "on the fly") as a query result (except for the predefined structured types); the type must first be defined so that an appropriate "create" operation exists (general facilities of this type are, however, in the area of research rather than practice). SQL3 is somewhat more limited in this respect, given the current lack of integration of the ADT-oriented and table-oriented facilities. It is still only possible to return tables from SQL3 queries, even though those tables can contain ADT instances (including structured ADT types) and some forms of nested literal structures.

4. Operating on user defined classes of objects.

Both ODMG and SQL3 provide these facilities.

5. Operating on other kinds of collections as well as sets.

Both ODMG and SQL3 allow queries to range over, e.g., multisets and lists. The collections to be queried can also be formed by nested query expressions.

6. Allowing the use of procedurally-specified operations in the query predicate.

Both ODMG and SQL3 allow this. In SQL3, ADT operations must be invoked on ADTs accessed via columns in tables.

7. Making use of encapsulation, inheritance, or operation overloading

Both ODMG and SQL3 allow queries to access only object (ADT) characteristics defined at object interfaces (i.e., queries do not break object/ADT encapsulation). SQL3 only provides encapsulation for tables to the extent that views are considered as providing encapsulation. ODMG supports standard object type inheritance and operation overloading. SQL3 supports this for ADTs, and supports a separate table inheritance concept with the ability to define overloaded operators.

8. Opening the query optimizer to allow new kinds of indices or new query algebras.

Neither ODMG nor SQL3 addresses this directly. Additional pragmas could be defined in ODMG to address some of these issues. SQL3 facilities for defining ADT implementations (e.g., for defining EQUALS and LESS THAN operations for ADTs) could be used to integrate additional kinds of indices in some cases. As noted already, the Illustra DBMS [Mon93] provides some examples of facilities of this type.

5.4.1.10 Formal foundations

SQL92 has the formal foundation provided by the relational algebra, and the associated concept of "relational completeness". However, this foundation does not apply to many of the extensions provided in SQL3. In particular, a formal foundation for the object extensions remains to be defined.

ODMG (and in particular OQL) currently has no algebraic foundation as an explicit part of its specifications, and thus no "completeness" property corresponding to "relational completeness" (although one could presumably be defined for its relational subset). The O_2 query facility is reported to be the basis of OQL, and a considerable amount of work has been done on the formal foundations of this facility, e.g., [AK89, BK86, CDLR89]. Such work, possibly extended with other work on object query algebras, e.g. [MD89, SZ89b], could be the basis for such a foundation. There is reported to be interest within ODMG in such work.

5.4.2 Merging the Standards

Both ODMG and SQL3 provide object models that include both atomic and structured literals and objects. Both also provide object query facilities that are similar in many respects. Strictly as an object-oriented query facility, ODMG is currently in many respects the more complete and better developed facility, since it starts from an object-oriented base, and thus is not restricted by a relational "legacy". It is also tightly coupled with IDL, and OMG facilities in general. ODMG really attempts (without saying so) to provide a superset of the relational model (using structured literals), in addition to its object capabilities. At the same time, ODMG currently lacks object facilities corresponding to some of the other facilities provided by SQL3, such as views, access control, etc. (i.e., although ODBMS products may provide such facilities, they are not currently part of the standard). Again strictly as an object-oriented query facility, SQL3 is currently somewhat less complete, and requires additional work. This is not unreasonable, given that SQL was not originally object-oriented, and the SQL3 developers have many other facilities under consideration in addition to those involving objects. It should be possible to define a mapping from SQL3 to a subset of ODMG facilities (as noted in Section 5.1, Rick Cattell, the ODMG Chair, has suggested that an SQL3 language binding could be made to ODMG-93 [Cat94c]).

Joint work between OMG, ODMG, and X3H2 should be encouraged to address harmonization of their object models (and syntax, where that is necessary). At the same time, both ODMG and SQL3 appear to have something to offer each other. Joint work should address the issue of defining cross mappings, and pinning down fuzzy areas that exist in both sets of specifications. Specific aspects of ODMG specifications that should be considered in SQL3 include:

- Objects can have separate persistent existence in ODMG, with structures such as tuples of objects using references to those objects (oids). SQL3 should consider this approach, instead of having separate reference and embedding concepts (embedding semantics could be defined by triggers if necessary). This would simplify the integration of objects into SQL3, and allow them to more closely resemble objects in other object-oriented systems. In particular, the facility in SQL3 of allowing an object to be embedded in one location, and still be directly referenceable from a different location, should be reconsidered. (SQL3's row identifier facility, and [Kel94c], could be considered steps in this general direction.)

- ODMG uses a classical object model. SQL3 should consider replacing its generalized model with a classical model. It is not clear how helpful the generalized model of SQL3 will actually be in defining object operations. Also, relatively few of the object-oriented programming languages that might be used with SQL3 use generalized models. It might be easier for SQL3 to start with a classical model now, and possibly add the generalized facilities later as they prove necessary or useful.

- ODMG fully integrates structured literals and objects. Maintaining the separation of table and ADT facilities as currently defined in SQL3 would be unfortunate.

- ODMG defines tightly-coupled bindings between its specifications and object-oriented programming languages. Similar bindings should be considered in SQL3, e.g., since the same sorts of client/server architectures that OODBMSs use now for supporting their object-oriented programming language interfaces might be useful in future Object/Relational DBMSs supporting SQL3.

ODMG has already clearly borrowed some things from SQL, including both aspects of OQL's SQL-like syntax (and, to some extent, semantics), and many of its built-in data types. Specific additional aspects of SQL3 specifications that should be considered by ODMG include:

- template types

- SQL3's more complete (or, at least, more completely specified) VALUE ADT facilities

- SQL3's facilities for views, access control, triggers (possibly using the Events Service), etc.

- SQL3's procedure and module facilities

- facilities defined by SQL3 that are essentially implementation-oriented, but might be useful in defining aspects of the OMG Query Service (e.g., the ability to override key operations such as EQUAL).

- specific syntax for the "relational subset" of ODMG structures

- user-defined CAST operations

- SQL3's bindings to non-object-oriented programming languages

Additional work could also be done to address the formal foundations of the SQL3 object extensions, and a possible object algebra for ODMG's OQL. This would, among other things, be helpful in defining query optimizers for these object query languages (see Section 6.4). The current work on formal foundations of object query facilities cited earlier has already shown that the mathematical theory of the relational model can be applied (with suitable extensions) to ODBMSs.

The ODBMS products described in Sections 3 and 4 illustrate varying degrees of support for object query facilities that also incorporate SQL-like relational capabilities. In some cases, e.g., Objectivity/DB's SQL++, the query facilities fully support the current SQL standard. On the other hand, the Illustra ORDBMS supports an earlier version of the SQL3 facilities. These products clearly indicate the merger of object and relational query facilities in actual products. As OODBMSs move toward supporting full relational query facilities, and relational DBMSs move toward supporting object facilities, similar technical problems must be overcome.

Commercial realities will create the need for most ODBMSs to support whatever the current SQL standard is (whatever its facilities are), or provide efficient gateways to other DBMSs that do support it, independently of any other query facilities they might support. At the same time, commercial realities will also create the need for DBMSs in general, either ODBMSs or relational DBMSs, to be easily integrated into distributed object architectures, most likely based on OMG specifications. The ODMG specifications provide one illustration of how database concepts can be integrated into the OMG framework. If these commercial realities are not addressed by actually merging the SQL3 and ODMG specifications, they will be addressed by providing a tight language mapping or binding between them. The common features of these standards described here should hopefully facilitate this process.

6. Key ODBMS Features and Issues in Providing Them

Sections 3 and 4 have described a number of different commercially-available ODBMS products. Section 5 has also, in a sense, described ODBMS products, in this case the future products (or components of them) that will be implemented to conform to the ODMG, SQL3, or OMG standards. This section will briefly review some key features found in ODBMS products (with emphasis on those features typically *not* found in relational DBMSs), describing some of the choices exhibited by the products in providing those features, and pointing out some of the technical issues in providing the features that sometimes come up in discussions of different ODBMSs (for example, some of the issues described below are sometimes cited in marketing literature as distinguishing one ODBMS from another). In general, these features are just as important for ORDBMSs as they are for OODBMSs, and the technical issues involved in supporting them are just as complex in the one class of system as the other. In addition to including material from [Man89a], this section includes some technical material derived from [CFZ94, DFMV90, Vel93]. A thorough discussion of ODBMS implementation issues and options can be found in [Cat94b].

6.1 ODBMS Persistence Model and Its Support

As noted in Section 1, DBMSs exist to provide persistent storage. *Persistence* is a property of data that determines how long it should remain in existence as far as the computer system is concerned. With most programming languages, data and objects in the program's address space are *transient*, i.e., they exist only as long as the program executes. Some data, such as locally declared data or procedure parameters, has an even shorter lifetime (the execution of the individual block or procedure).

In a conventional programming language, to make an object *persistent* (exist beyond the lifetime of a single program execution), the programmer must include explicit instructions to save the object on persistent storage, such as in a file on disk (possibly using a DBMS as an intermediary). Conversely, to reuse an object that was created in an earlier execution, or by some other program, the programmer must include explicit instructions to fetch the object from persistent storage (e.g., from the DBMS). Section 1 noted some of the problems with this approach, in particular, some of the performance advantages of having a closer correspondence between programming language and persistent (DBMS) types. Basically these advantages involve minimizing the processing required to translate between application program objects and database objects. This can be significant in many advanced applications when a conventional relational DBMS is used.

Ideally [AP87], persistence should be:

- *orthogonal* (to types): any data of any type can persist at any time
- *transparent*: the programmer can treat persistent and transient (non-persistent) objects the same way
- *independent* (from secondary storage): there should be no explicit reads and writes into the database

Different products achieve these goals to different extents, and none achieve them perfectly. The way in which an ODBMS presents persistence to users is sometimes termed its *persistence model*.

A *seamless* persistence model is one in which the ODBMS's model or type system is a persistent extension of that of one or more host programming languages, as in the ideal just cited. Section 1 noted some of the advantages of such a model in program development, maintenance, and reuse. However, there are also a number of problems with attempting to provide seamlessness. First, there are problems in trying to be completely seamless with languages that do not have the proper concepts (e.g., versions, clusters, indexes, transactions, persistence, or, in some cases, even objects) for dealing with the DBMS. For example, if a language has no notion of database transactions, and it is important to the application to be able to use this concept, it is necessary to provide some way for the application developer to express this concept within the language (either by special syntax or by use of special subroutine calls). Second, seamlessness is more difficult if multiple programming languages are to be supported by the DBMS, since these languages may have different type systems, models of data sharing, etc., and it may therefore be necessary to allow programmers to be involved in dealing with the necessary conversions. Finally, of course, the underlying mechanisms providing key DBMS facilities (e.g., moving data between persistent and transient storage) must be implemented in an efficient way. The need for efficiency sometimes involves tradeoffs in how seamless the mechanism actually is. In addition, partially due to the way that relational DBMS interfaces have historically been designed, OODBMSs and ORDBMSs sometimes have different goals in terms of providing more or less seamlessness. Persistence models can differ from one ODBMS to another. Moreover, the same ODBMS may present different persistence models at different language interfaces, in order to align the persistence model more closely with the characteristics of the programming language involved.

There are a number of different aspects of an ODBMS persistence model. Several of these are discussed in the following subsections.

6.1.1 How Is The ODBMS Informed of Object Persistence?

One aspect of an ODBMS's persistence model is *how the ODBMS is told that a given object is to be persistent* (or its inverse, how the ODBMS is told that an object need no longer be persistent and can be deleted). Telling an ODBMS that an object is to be persistent may not necessarily be straightforward if the ODBMS attempts to provide a seamless interface, since there may no longer be an explicit "write" operation to a separate "database" that explicitly indicates this.

Perhaps the simplest, and most familiar, model is sometimes termed *database schema declaration*. This is the model used by conventional relational DBMSs. This model is also used by ITASCA, MATISSE, and the ORDBMSs, and to some extent by GemStone (for Smalltalk DB objects). In this model, the ODBMS provides the appearance of a separate space (the database) of persistent objects. Persistent classes are declared in the ODBMS, and all instances of those classes are persistent. Applications operate with a separate set of transient classes in their memory. In this model, objects are defined as persistent by explicitly creating them in the database, and are generally made non-persistent by deleting them from the database (although GemStone uses garbage collection, as described below). Translation between persistent ODBMS objects and transient programming language objects must generally be done by the programmer (although some systems provide assistance in automating this), effectively by moving object state between persistent and transient objects. This is similar to the interface between programming languages and conventional relational DBMSs, except that the type systems of the programming language and an ODBMS may more closely resemble each other. If the ODBMS implements methods in the database (see Section 6.2.2), references or handles to database objects can also be obtained by application objects, and messages sent from the application to the database object to invoke the execution of database object methods.

The most frequently-used model in OODBMSs is that of *explicit persistent instantiation*. This is used, e.g., by ObjectStore, Objectivity, VERSANT, ONTOS, and GemStone in their C++ interfaces, and in the ODMG C++ binding. In this model, objects are made persistent by explicitly designating them as persistent (and deleted by explicitly deleting them). This is either done on a per-instance basis for objects of any class (ObjectStore), or on a class basis. When this is done on a class basis, specific classes are indicated as having persistent instances (e.g., by defining them as subclasses of some predefined database object class); both persistent and transient instances of these classes can then be created (e.g., Objectivity, VERSANT, and ONTOS). Unlike the previous model, the object classes in persistent and transient spaces overlap, since there are classes that have both persistent and transient instances.

Referential integrity (ensuring that, in a reference from one object to another, the object referred to actually exists) is the responsibility of the programmer (modeling the characteristics of C++). Referential integrity can be an issue in these systems both because objects can be explicitly deleted, and because an application can create a reference between a persistent object and a transient object in its memory, and fail to make the transient object persistent when the persistent object is written back to the database. (In fact, it may be impossible to make the referenced object persistent because it may not be of a persistent class). Many of the products provide a capability for defining inverse relationships that can be used to check referential integrity if this is desired.

The need to identify specific classes as being persistent means that, when such classes are defined in a programming language independently of the ODBMS (e.g., in a typical C++ interface), an explicit mechanism must be provided to transfer those definitions from the programming language to the ODBMS. This is typically performed by a special utility (called *classify* in ONTOS, or the *registrar* in GemStone), as illustrated in Figure 6.1.1.

A third model is *reachability from persistent roots*. This model is used by O_2, the Smalltalk interfaces of a number of products (e.g., ObjectStore and VERSANT), and the Smalltalk binding in the ODMG specifications. In this model, an object is made persistent when it is referred to by another persistent object. These systems generally provide built-in referential integrity, both because they guarantee that if an application creates a reference between a persistent and a transient object, the transient object will also be made persistent, and because they generally provide *garbage-collection* semantics. In garbage collection, unneeded objects are deleted automatically, rather than allowing (or requiring) users to explicitly delete objects when they are no longer needed Objects are garbage-collected when no references to the object exist in other system-maintained objects. This approach has the advantage that the programmer need not worry about doing this explicitly, or determining the correct conditions under which an object is no longer needed. This also guarantees that references to an object that exist in other objects are always valid, since the fact that a reference exists assures that the object exists. This can be an important advantage when objects with complex interrelationships with other objects need to be maintained.

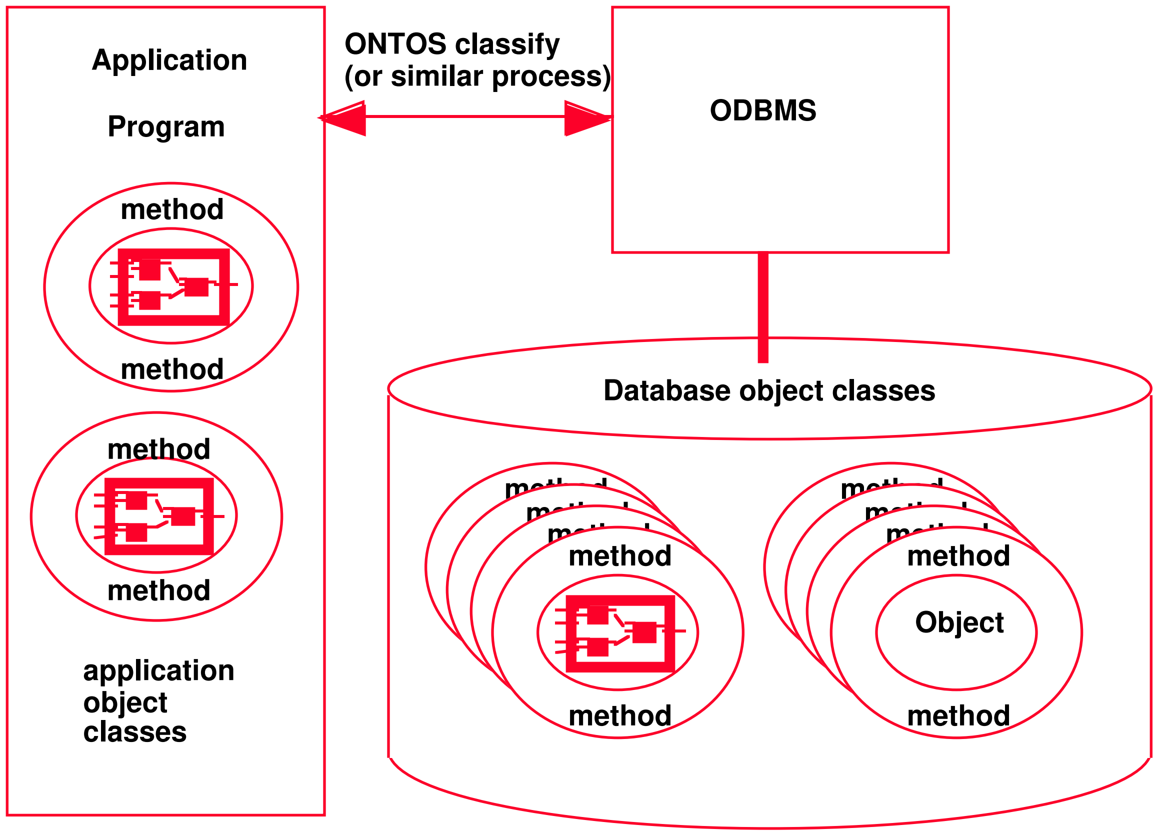

Figure 6.1.1 Registering Database Classes

The alternative of allowing objects to be explicitly deleted makes the enforcement of the more complex forms of referential integrity extremely difficult. Systems that do not support such integrity can attempt to "fix" dangling references by removing them or taking some other corrective action. Such actions almost always involve finding the objects containing dangling references or recording all references to an object. Thus, from a semantic point of view, explicit deletion can be extremely dangerous. From a performance point of view, there may be no inherent advantage to fixing dangling references over garbage collection, depending on the efficiency of the alternative used.

The reachability model is sometimes combined with explicit instantiation. For example, VERSANT allows a Smalltalk object to be explicitly made persistent by sending it a message to make itself persistent. In this case, the ODBMS automatically accesses any Smalltalk class information necessary to define the ODBMS class or type corresponding to the instance.

6.1.2 How Does an Application Access Persistent Objects?

Another aspect of an ODBMS persistence model is *how an application accesses persistent objects*, i.e., how persistent objects are brought into memory when required, how persistent structures are traversed by the program, how transparent this is, and what mechanisms are required to support this access. Again, this can be more or less straightforward, depending on how seamless the ODBMS attempts to make this.

In the case of the ODBMSs that maintain an entirely separate space of persistent objects and persistent types or classes (e.g., ITASCA, the ORDBMSs, the C interface to Smalltalk DB objects in GemStone), dealing with this issue is straightforward. Two approaches are possible. One approach is for an application to access a persistent object by explicitly reading its state to a transient object or data structure, and then, if the object is to be updated, operating on that structure in transient space. Once any manipulation is complete, the transient data is then used to explicitly update the corresponding object in persistent space. The other approach is for the application to merely obtain references to persistent objects, and to send messages to the persistent objects to invoke operations that are performed entirely in persistent space. This may be done at the granularity of individual objects, or by sending queries to the ODBMS (the primary mode in the ORDBMSs). In both these approaches, the application is fully aware of the distinction between transient and persistent objects, and the type systems of persistent and transient objects are generally different.

However, many of the ODBMSs attempt to provide a more seamless interface, in which an application can largely ignore whether the objects it is accessing are persistent or transient. For example, much of the access in an OODBMS is accessing an object (e.g., invoking one of its operations) via a reference to it by an attribute in another object already known to the application. An example of this type of access is, given an `Employee` object, the application may wish to evaluate the `age` operation on the object referenced by the `Employee` object's `Manager` attribute (in a typical notation this would be evaluating the expression `X.manager.age`, where `X` is the `Employee` object originally known to the application). Similarly, given a `Gate` object in a VLSI design, the application may wish to find the next gate along some signal path by finding the `gate_type` of the object referenced by the `output_pin` attribute of the original `Gate` object `X` (evaluating `X.output_pin.gate_type`). This traversal of the object structure by following references to related objects is sometimes called *pointer-chasing* (since the object references involved are effectively pointers, and are frequently translated into genuine pointers in the application programming language).

Suppose the application has an object in its memory which contains a reference to another object located in the database. This situation is shown in Figure 6.1.2 (taken from ONTOS product literature). The left side of the figure shows a collection of objects in an object database, and the right side the corresponding in-memory structure created by bringing some of these objects into the application's memory. On the left, the shaded circles show objects reachable from object X (shown in black). The question is, what happens (in the application program on the right) when the application attempts to follow (dereferences) the pointer which points to the question mark (e.g., invokes a method of the object pointed to)? Specifically:

a. does the application need to know that the object is not in memory, and bring it in before dereferencing the pointer, or, if the application does not need to do this, how does the ODBMS find out the object is not in memory and bring it in?

b. what type of translation needs to be done when moving the object from persistent storage into main memory (and vice-versa) and who does it?

c. does it make sense to bring in more than one object in this case, i.e., objects other than the one explicitly referenced?

Question (a) is essentially the issue of what the interface looks like to the application, and what tradeoffs exist in providing that interface. Question (b) is essentially the issue of how the interface is implemented. Question (c) involves the particular implementation concept of *clustering*.

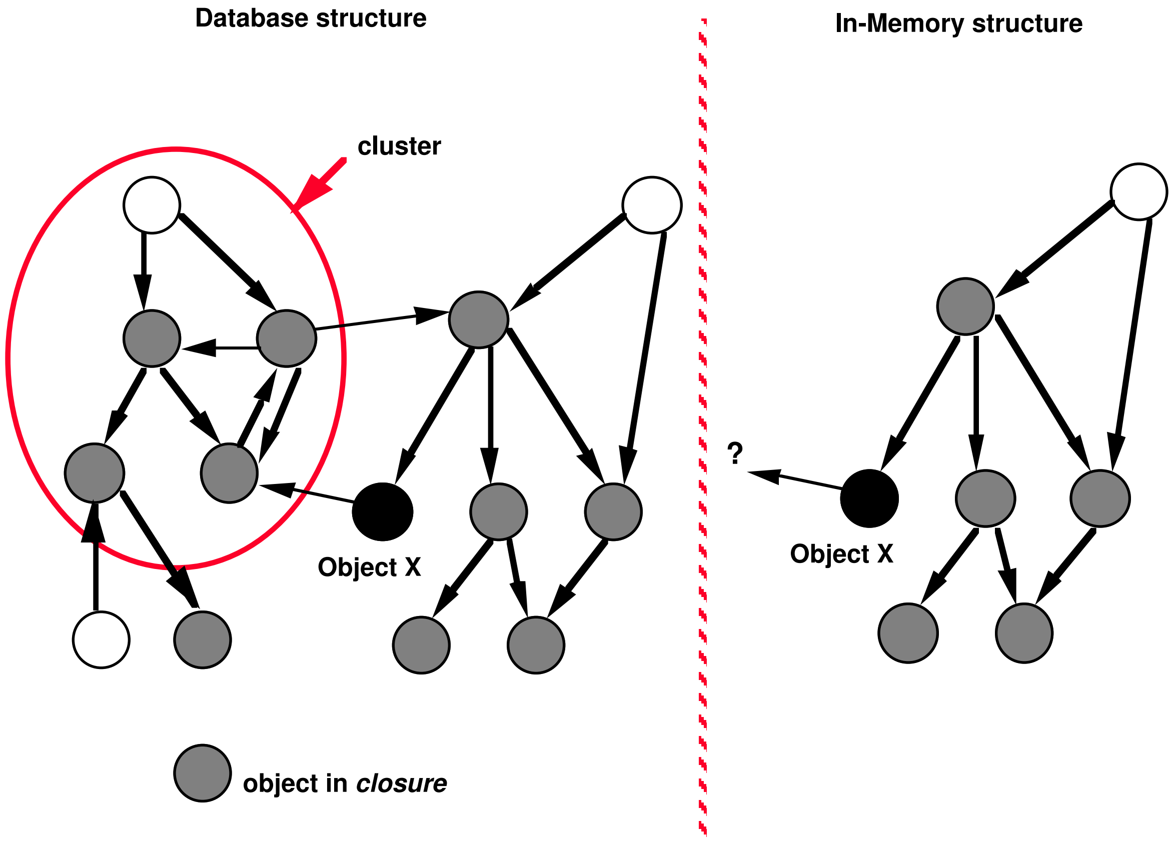

Figure 6.1.2 What Happens Next?

In addressing the issue of what the interface looks like to the application, ideally, the program would follow that reference (dereference the pointer) as if it were a pointer to another object in memory, and the ODBMS (the client part located with the application) would automatically detect that the object was not in memory, read the object from disk into the application's main memory, and the application would proceed. How transparent this process actually is depends on how pointers (oids) are implemented in the ODBMS and in ODBMS applications (or if there are multiple options, which option has been selected by the user).

Most ODBMSs use separate database and application pointer formats, with some translation mechanism employed when an object is moved from the database into the application's main memory. This is because there are different requirements for main memory and disk-based pointers. Pointers to objects in programming languages are typically implemented by virtual memory addresses. For persistent objects, pointers that are not dependent on the object's physical position must be used, since there is no guarantee that the position will remain constant between uses by different processes. Instead, some kind of system-generated object identifier (e.g., a GemStone *oop* or an ONTOS *OID*) is generally used. Dereferencing these pointers usually requires looking up the value in a table to locate the object on disk. Once the object is located and read into virtual memory, it is desirable to eliminate the overhead inherent in this table lookup by replacing any in-memory references to the object with the object's new virtual memory address. Even if object identifiers that must be looked up in a table, rather than virtual memory addresses, are used in the programming language (as they have been in some implementations of object programming languages) as well as on the disk, the *form* of the object identifiers may need to change when they are moved between disk and main memory. For example, the object space on disk may be larger than that which can be supported by the programming environment. In this case, the disk-based identifiers must be large to uniquely identify the required number of objects, while the memory-based identifiers may need to be much smaller in order to conserve the use of memory by objects made up of many pointers. Various schemes have been proposed for temporarily replacing the disk-based pointer with a main-memory pointer and then swapping it back to the disk-based form when the object is returned to the disk (see, e.g., [AP87, MSOP86, Kho87, Kae86]). This conversion between the persistent form of a pointer and the virtual memory form is known as *pointer-swizzling*.

The ONTOS C++ interface illustrates many of the issues involved in question (a) above. In ONTOS, 64-bit object identifiers (OIDs) are used in the database for inter-object references. Objects must be moved from the database into the application's memory in order to execute object methods, since in C++ the method code is linked into the application. As described in Section 3.2, objects are transferred between the application's memory and the database by operations called *activation* and *deactivation*. Activation involves transferring object state from the database to memory. During activation, all references contained by the activated object to other already-read objects are translated (swizzled) from their OID form to virtual memory references. Also, all the references to the newly-activated object from other already-read objects are similarly translated. Deactivation is the reverse process: the translation of memory references back into OIDs, and the writing of objects back to the database.

The form of virtual memory reference used when object state is moved from database to application memory can be controlled by the programmer in ONTOS, using specifications in object class definitions. When *abstract references* (instances of class `OC_Reference`) are used, all references go through one level of indirection, and are guaranteed safe by the system. If an object referenced by an abstract reference is inactive (still on disk), this is detected during the process of dereferencing the abstract reference, and the referenced object is activated automatically by the system, using information associated with the abstract reference to locate the object on disk. Activation is thus transparent to the programmer, at the cost of the additional indirection. Abstract references can be used to guarantee reference safety when individual objects are activated. However, if the application can determine valid references itself, *direct references* may be used. In this case, when an object is activated, references to objects already in virtual memory are translated to standard C++ virtual memory pointers. Direct references are used directly, without any checking, and have the same performance as ordinary C++ references, but the application must insure that it does not dereference an untranslated reference. If the application detects the presence of an direct reference that has not been activated, the object may be activated explicitly by a function call. Objects may also be activated explicitly by name, using the `OC_lookup` operation.

Most of the ODBMSs described in this report use a separate format for database and in-memory object references, and provide facilities corresponding to either or both of ONTOS's direct and abstract references. For example, GemStone provides DPointers (which correspond to ONTOS direct references) and GPointers (which correspond to ONTOS abstract references). Objectivity/DB and VERSANT effectively provide only abstract references; that is, they do not allow the use of direct pointers. All references to persistent objects in these systems must be of a special persistent object reference class (e.g., class `Link` in VERSANT). These vendors argue that providing only indirect object references provides a measure of safety and database integrity for the application (it cannot get into trouble dereferencing an invalid pointer), as well as transparency, that outweighs the performance cost.

As noted in Section 3.3, ObjectStore uses an entirely different approach. ObjectStore provides a single-level storage model for applications, in which persistent objects are mapped into the program's virtual memory space. With this approach, objects in the database and in main memory have exactly the same format, and references both in the database and in main memory are always virtual memory pointers. Moreover, there is no explicit activation and deactivation; the ODBMS can detect when an object not in application memory is accessed via the virtual memory mechanism, and bring it in automatically, even though a virtual memory pointer rather than a special pointer type is being used. Thus, these pointers always behave like ONTOS abstract references as far as transparently accessing an object in the database is concerned.

However, even though it uses this virtual memory mapping approach as its basic mechanism for accessing persistent objects, ObjectStore also provides additional pointer types, similar to ONTOS's abstract references, for use in various special cases when the virtual memory pointers cannot be conveniently used. Specifically, ObjectStore provides special enhanced pointers called *references* (class `os_Reference`, and its subclasses). These references can be used as cross-database pointers, or as pointers that remain valid across transaction boundaries, at the cost of some additional dereferencing overhead and pointer space.

An issue related to deactivation occurs when objects are updated. In an ODBMS supporting a seamless programming language interface, when an object has been activated (brought into memory) and operated on by the application, it may not necessarily be obvious to the ODBMS whether the object has been modified or not. Thus, when a transaction involving the object commits, it may not be clear whether the object must be rewritten back to the database or not. This is because, as opposed to the situation in a relational DBMS when an explicit UPDATE operation to the database is invoked, the operations that an application invokes on an object may invoke methods that are linked into the application, and which give no explicit indication to the ODBMS as to whether they involve reading or writing the object state (see Section 6.3.1). Some ODBMSs, e.g., ONTOS and VERSANT, provide an explicit operation to mark an object as modified which must be invoked within object methods that perform updates. ObjectStore, due to its use of the virtual memory mechanism to handle object mapping, can automatically detect when an object has been modified.

Question (c) above referred to the issue of whether to bring more than one object into memory when a given object was referenced. This is an issue because one way of improving performance in a situation where object graph structures, such as the one in Figure 6.1.2, are being traversed is to minimize the probability that traversing an edge in the graph will cause a disk fault due to the referenced object not being in main memory. This can be done by *intelligent prefetching*, to ensure that objects likely to be accessed during the traversal are already in main memory when they are accessed. This makes *clustering* (physically grouping objects likely to be accessed together so they can be moved as a unit) a very important issue in object-oriented systems. Issues include how to cluster object data so that appropriate parts of the object are staged to match the operations to be performed (the entire object may not be needed, and may be too large to fit in an application's memory in any case), and how to detect when related objects have to be moved along with another one, which may depend on the operations that have to be performed on the moved object.

Many ODBMSs provide some mechanism for defining clustering. In current products, these mechanisms are *static*; i.e., an object is assigned to a cluster when it is created, and any change in clustering requires an offline reorganization. A number of different ways of assigning objects to clusters are used in ODBMS products. Some products have defined physical clustering units (e.g., *cluster* in ONTOS, *segment* in GemStone) in which applications can explicitly place objects to indicate clustering. O_2 provides the ability to cluster hierarchical groups of objects based on internally-defined composition relationships. ITASCA allows clustering based on composition (IS_PART_OF) relationships defined at the schema level.

Clustering is not without its problems, however. In static clustering, only one class of applications can be optimized for disk access. This creates a problem for the database administrator in trying to adapt the clustering organization to changing workloads. An additional issue is that while prefetching is frequently considered as a possible optimization in object-oriented systems, prefetching is not cost-free. It only provides significant benefit if the hit rate is relatively high, since otherwise, the cost of moving unnecessary data will outweigh any efficiencies that might be achieved. Even when an ODBMS uses its own data language, and thus might be capable of analyzing its own method code during compilation to discover prefetching opportunities, a high hit rate for prefetched data is not necessarily guaranteed. In fast workstations, a great deal of computation can be carried out in the time required to fetch an object from disk (particularly if this involves access to a remote server). This makes it more difficult to determine what to fetch, with a high probability of being correct, far enough ahead of time to realize any advantage.

In addition, as noted earlier, something akin to the *inverse* of clustering must also be addressed in some ODBMS applications. If objects can be of arbitrary size, they may be larger than the available main memory (e.g., 40 megabyte images are not infrequent in some applications). In this case, it must be possible to dereference a pointer to the object and *not* bring the entire object into main memory. In general, applications need some control over how much of an object is brought into memory in these cases. Some variant of the operations for handling BLOBs in relational systems (e.g., those supported by Illustra, as described in Section 4.2) are often provided.

6.2 ODBMS Architectures

OODBMSs generally have architectures that are fundamentally different than those of conventional relational DBMSs (RDBMSs). The nature of this difference is illustrated by Figure 6.2.1 [Vel93]. As shown in the figure, both the RDBMS and the OODBMS have client/server architectures (components of the DBMS are shaded). The difference is in the division of labor between client and server. In the conventional relational system, the architecture is designed for applications to send SQL queries to the server. All query processing is done in the server, which returns rows satisfying the query to the client. The server also handles transaction processing, recovery, etc. The client's responsibility is to handle application processing, together with managing cursors that allow the application to range over the rows returned by the query. This architecture avoids a lot of network traffic, because only rows explicitly required by the application are returned from the server. This architecture is also a good idea if the server is extremely powerful compared to the clients (e.g., when servers are mainframes, and clients are the earlier classes of personal computers with relatively little memory or computational power).

On the other hand, in the typical OODBMS architecture, the client plays a much greater role. These architectures are based on what is sometimes called a *data-shipping* approach [CFZ94]. In this approach, data is shipped from servers to clients so that much of the OODBMS's processing (sometimes including query processing), as well as application processing, is performed at the client. This provides two advantages. The first is that the interfaces provided by OODBMSs generally emphasize navigation through the complex graph-like structures illustrated in Sections 1 and 2, requiring that the application follow direct references from one object to another (pointer-chasing). The data-shipping approach moves the data closer to the application, and supports efficient fine-grained interleaving between application processing and access to the next persistent object (found by following a pointer) required by the application. The second advantage is that DBMS functions are off-loaded from the server to the clients. This allows a given server to support many more clients, and takes maximum advantage of the resources of the (typically numerous) clients in the network. The difference in architecture creates an enormous performance advantage for OODBMSs over relational DBMSs in many OODBMS applications (see, e.g., [CS92, CDN93, CFZ94])[15].

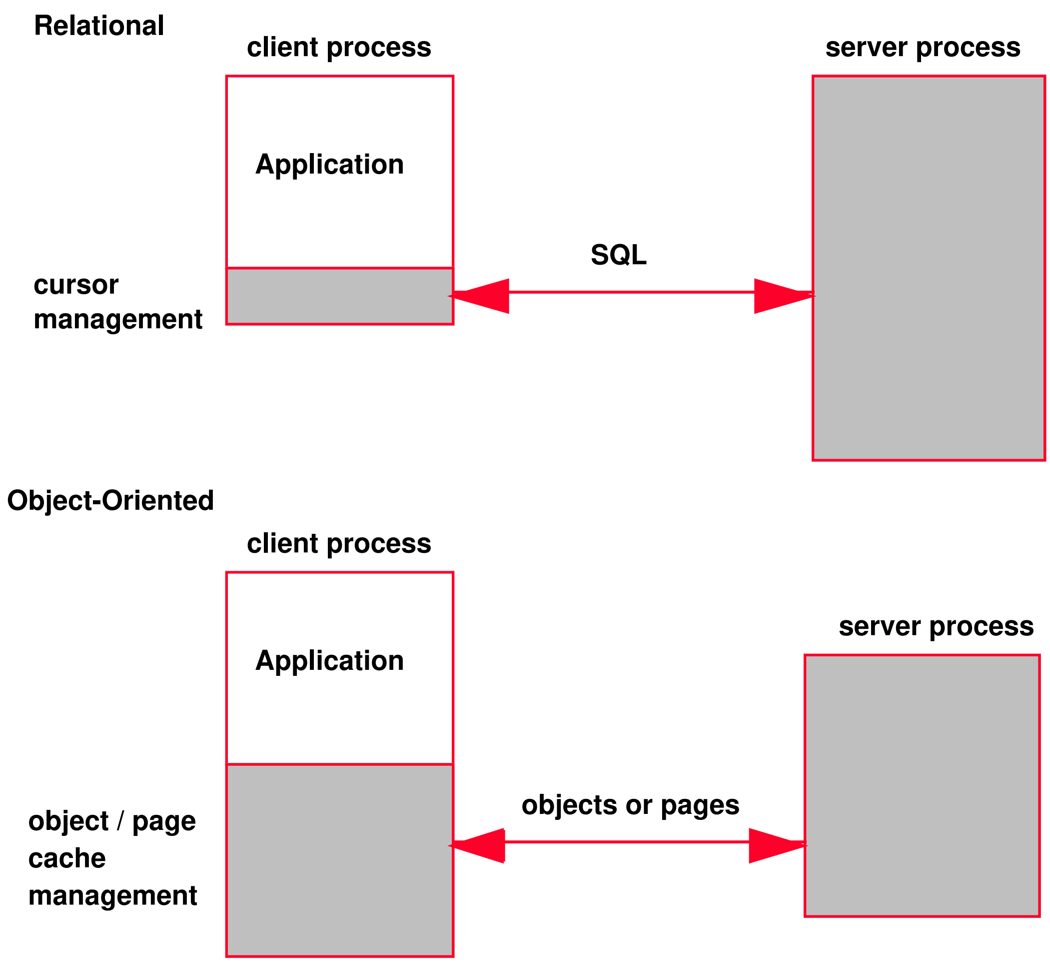

Figure 6.2.1 DBMS Client / Server Architectures

---

[15] In particular, [CS92] and [CDN93] describe key OODBMS performance benchmarks.

Even when a data shipping approach is used, variations exist in how it is implemented. As noted in [CFZ94], ODBMS applications operate logically, in terms of objects. The objects themselves, however, are stored on disks in storage units (*pages*). Thus an ODBMS must at some level manage storage in terms of data pages, and also manage the mapping between objects and pages. However, a client/server architecture per se does not dictate where in the system the mapping between objects and pages must take place. As a result, existing ODBMSs differ in the granularity at which their clients and servers interact. There are two basic approaches: *page servers*, where clients and servers interact in terms of pages (or groups of pages), and *object servers*, where clients and servers interact in terms of individual objects. The choice between these approaches has a significant effect on the design of many ODBMS functions, such as concurrency control, recovery, and query processing. While this choice may seem to be primarily of interest to ODBMS implementors, a considerable amount of discussion in sales literature and trade press articles focuses on ramifications of the page server/object server distinction (see, e.g., [Bar94b, Loo92,94]). Conventional RDBMSs can be factored into this categorization by defining them as systems in which the unit of transfer between the client and server is neither the object nor the page, but rather the *query*. Thus, these systems might be termed *query servers*. Of the ORDBMSs, both Illustra and OpenODB are query servers, while UniSQL appears to operate as either an object server or query server. Query servers often provide some amount of flexibility in assigning non-query work to the client or server. For example, most RDBMSs provide the ability to define stored procedures that execute on the server. Conversely, the Illustra ORDBMS provides the ability to optionally specify that an ADT function (object method) is to execute on the client, rather than on the server as functions in Illustra normally would.

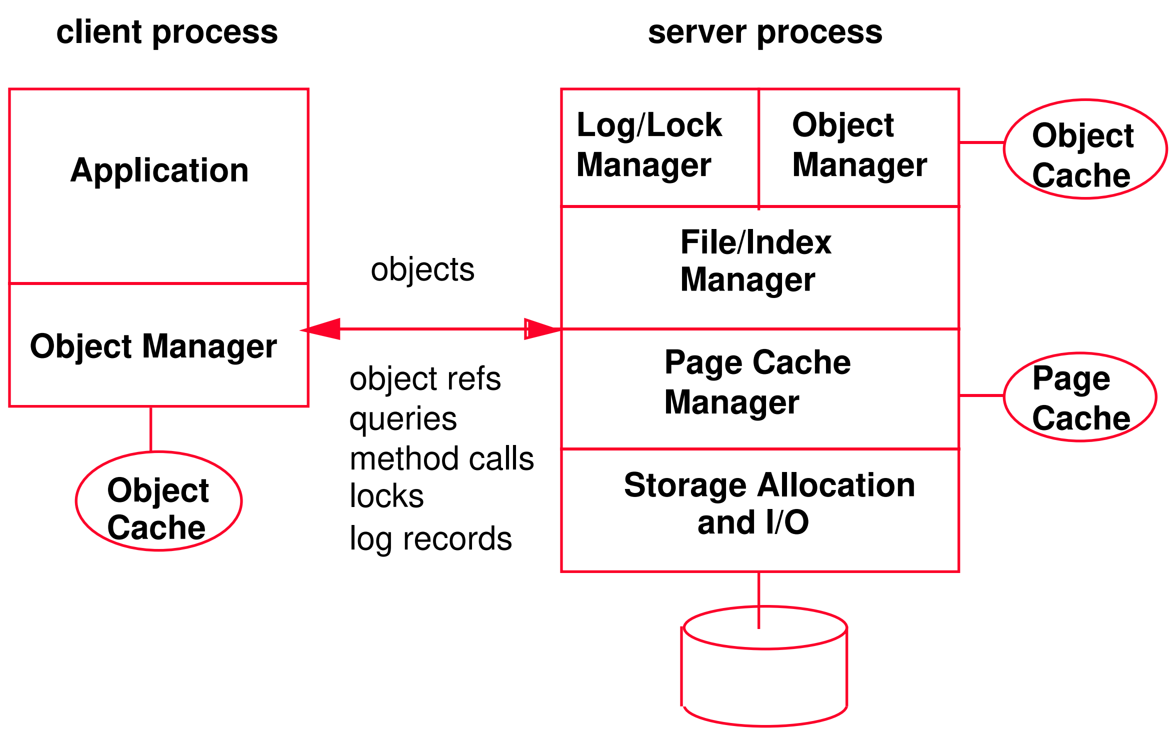

Figure 6.2.2 Object Server Architecture

Figure 6.2.2 shows a possible configuration for an object server[16]. In the object server architecture, the basic unit of transfer between the client and the server is the object, although other information can also be transferred, as shown in the figure. This means that both the client and server must have an object manager component that can manipulate individual objects. Both client and server maintain object caches of recently-accessed objects. When the client needs an object it first searches its local object cache. If the object is not found, it sends a request for the object to the server. If the object is not in the server's object cache, the server (in this particular configuration) first checks a page cache of recently-accessed disk pages. If the object is in one of these pages, it is returned to the client. Otherwise, the server retrieves the object from disk and returns it to the client. The server may or may not keep a copy of that object in its local object cache, depending on the cache policy it implements.

VERSANT, ITASCA, and the ONTOS standard storage manager use the object server approach. This architecture has a number of advantages. First, because objects are recognized on both the client and server, it is possible to arrange for both the client and server to execute methods (although a given ODBMS may not provide this capability; also, this depends on whether the ODBMS actually stores object methods, a subject discussed in Section 6.3). Thus, a method (query) that selects a small subset of a very large collection could run on the server, and avoid moving a large amount of data to the client for processing. This can be important if there is no index on the collection being searched. This also allows performance to be tuned by moving processing between client and server. Moreover, since the server knows exactly which objects have been accessed by each application, concurrency control can largely be centralized in the server, and the implementation of object-level locking is straightforward, since the server is able to distinguish individual objects. Finally, only objects needed by the application are actually moved, and thus are at risk of being corrupted if the client or its software malfunctions (and even then, the server is in a position to validate some aspects of changed objects as they are written).

The object server architecture also has a number of disadvantages. First, in the worst case there may be a separate remote procedure call (RPC) to the server for each object reference, although hopefully most requests will be satisfied by objects in the client object cache. For the server to transfer more than a single object at a time, either the server must be able to figure out which objects belong with one another, replicating the work that any clustering mechanism did when it placed objects together on a disk page, or the application must be able to explicitly request that a particular group of objects is to be transferred (ONTOS, for example, allows this).

A related problem arises when a method is applied to a group of objects that are currently distributed among the client cache, the server cache, and the file subsystem (page cache and disk), since the same object may be in both object caches as well as in storage. Moreover, an updated version of an object may exist in the client object cache, but not in the server object cache. Thus, a method cannot simply be executed on the server without addressing these problems of cache consistency. O_2 Technology reported its experience with an initial implementation using an object server architecture (now replaced by a page server) as being that a significant overhead was imposed on the system to address these potential cache inconsistencies.

---

[16] Figures 6.2.2 through 6.2.4 are from [Vel93]; variants of them appear in [DFMV90], which is also the source of much of this discussion. These are *generic* architectures; not all ODBMSs have these specific identifiable components.

Another disadvantage is that the architecture complicates the design of the server. Instead of implementing only the facilities that it alone can provide (e.g., sharing, concurrency control, and recovery), the server in this design must duplicate the object management facilities of the client as well (possibly including the ability to execute object methods). An additional minor problem is that, since the client simply requests an object from the server without necessarily being aware of the size of the object, large multipage objects may be moved in their entirety even if the application needs to access only a few bytes, depending on the details of the implementation.

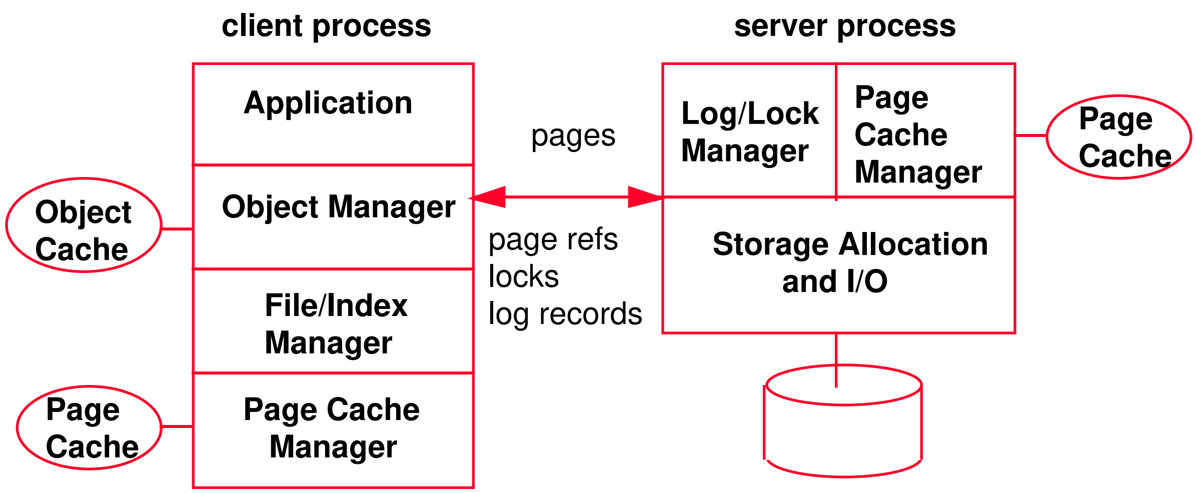

Figure 6.2.3 Page Server Architecture

Figure 6.2.3 shows a possible configuration for a page server. In the page server architecture, the server basically consists of a page cache (buffer pool), the I/O level of a storage system for managing pages and files, plus facilities for concurrency control and recovery. The upper levels of the ODBMS software run entirely on the client, and the unit of data transfer between the client and server is a disk page or group of pages. Caching on the client can be done in terms of pages, in terms of objects, or both (Figure 6.2.3 shows both types of caches). The advantage of an object cache is that cache space on the client is not wasted holding objects that have not actually been referenced by the application. However, if an object cache is used, then the client must copy objects into the object cache from incoming data pages, and if an object in the cache is updated, the page on which the object resides may have to be re-read from the server.

O_2, ObjectStore, GemStone, and the ONTOS group storage manager use the page server approach. Like the object server, the page server architecture also has both advantages and disadvantages. The main advantage of a page server architecture is that it places most of the complexity of the system in the client workstations, where the proponents of this architecture expect the majority of the available CPU cycles to be available, leaving the server to perform the tasks that it alone can perform (concurrency control and recovery). Since entire pages are transferred between the workstation and the server, the overhead on the server is minimized. While at first glance this approach may appear wasteful if only a single object on the page is needed, in fact the cost (in terms of CPU cycles) to send a 4K byte page is not much higher than the cost of sending a 100 byte object. In addition, if the clustering mechanism works properly, a significant fraction of the objects on each page will eventually end up being referenced by the client. Moreover, for a large, multipage object, the client can work with only those pages of the object actually needed. Finally, by minimizing the load each client places on the server, it is possible to support more clients using the same server.

One of the disadvantages of this approach is that methods can be evaluated only on the client. This means that certain kinds of processing, such as a sequential scan of a large collection, may be expensive, due to the large amount of data that may need to be moved between server and client. Another disadvantage is that object-level concurrency control is more complicated to implement, or, if page-level concurrency control is implemented, concurrency may be reduced. The performance of this architecture may be heavily dependent on the effectiveness of object clustering. Finally, pages cached on the client may be corrupted by a malfunctioning application, and may be written back to the server without the server being able to detect the corruption. These pages may include system information, such as indexes, in addition to user data.

There are also mitigating factors for most of these page server disadvantages. For example, in considering the problem of not being able to execute methods on the server, the server does the same amount of disk I/O when processing a long sequential scan as in the object server design. Second, in the case of an indexed selection, only those pages containing relevant objects will be moved from the server to the workstation (and only those pages of the index actually needed will be moved as well). Third, even an unindexed query on, e.g., a collection of images, would typically fetch only the page of each image where its identifying information is stored; the vast majority of pages which constitute the image itself would only be transferred to the client machine if actually requested. Also, the page server avoids the problems of cache consistency associated with the object server architecture when executing methods on the server. [CFZ94] reports on several new approaches for supporting object-level locking in page server architectures which provide improved performance over currently-used techniques (the tradeoffs between object-level and page-level locking are further discussed in Section 6.5). Finally, in cases where it is necessary to have methods performed at a server site (for example, if the server machine has image-processing hardware and the clients do not), some ODBMSs (e.g., ObjectStore) can be configured with a client application running at the server site. Such an application could implement the special methods, and act as a network server for the ultimate client applications [ODI93].

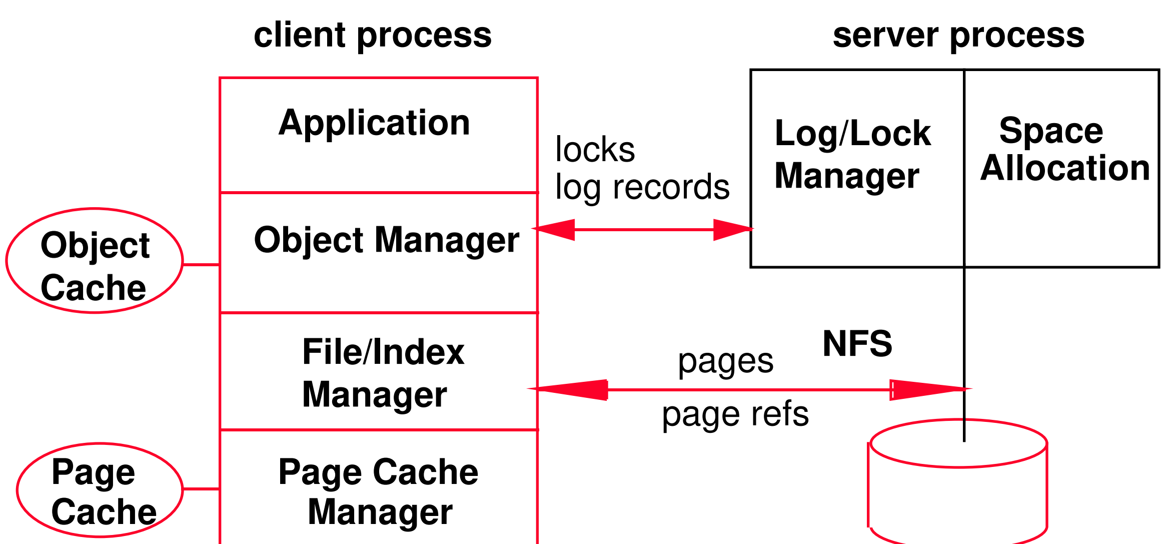

Figure 6.2.4 File Server Architecture

Figure 6.2.4 shows the *file server* architecture, which is a variation on the page server architecture in which the client uses Sun's NFS remote file service to read and write database pages directly. The file server architecture is used by Objectivity/DB. This architecture exhibits both all of the advantages and all of the disadvantages of the page server architecture described above. An additional advantage is that, since NFS runs in the operating system kernel, using it to read and write pages avoids user-level context switches, thus improving retrieval performance. A disadvantage is that NFS writes can be slow. Moreover, because page read operations bypass the server entirely, it is not possible to both request a page and request a lock on the page in the same message. The need to send separate messages may negate some of the performance advantages of using NFS in the first place. Alternatives are the use of optimistic concurrency control, or to batch lock requests to reduce overhead.

Evaluating the pros and cons of these different architectures (e.g., in ODBMS selection) requires detailed knowledge of application and hardware infrastructure characteristics and requirements, performance requirements, lifecycle costs and requirements, and many other implementation-oriented details. It also requires a detailed knowledge of the particular implementation choices made by the candidate ODBMSs (since there are different variants of these architectures, and ways to mitigate the various disadvantages cited often exist), as well as the ability to weigh these implementation choices against the other features provided by the various ODBMSs under consideration.

6.3 Support for Object Behavior

As noted in Section 2, ODBMSs provide persistent storage for objects, which incorporate their own behavior. As a result, ODBMSs must provide support for object behavior, i.e., the execution of object operations (methods). However, there are a number of different aspects of providing this support, and different ODBMSs support object behavior in different ways. The following subsections discuss these issues.

6.3.1 ODBMS Execution of Object Behavior

As originally described in Section 2.2, an ODBMS is a DBMS that stores not data, but *objects*: encapsulated combinations of data structures together with associated procedures (methods). This concept, shown in Figure 6.3.1, generally conveys the impression that complete objects (i.e., both object state and object methods) reside in the ODBMS, and also that objects "behave" in the ODBMS, i.e., that methods execute in the ODBMS. However, one of the issues involved in ODBMS support for object behavior is whether the ODBMS itself actually stores and executes object behavior, or whether object behavior is actually stored with the application program, and executed as part of the application program's execution environment (for example, [Loo94] describes it as a "myth" that ODBMS's [generally] store objects).

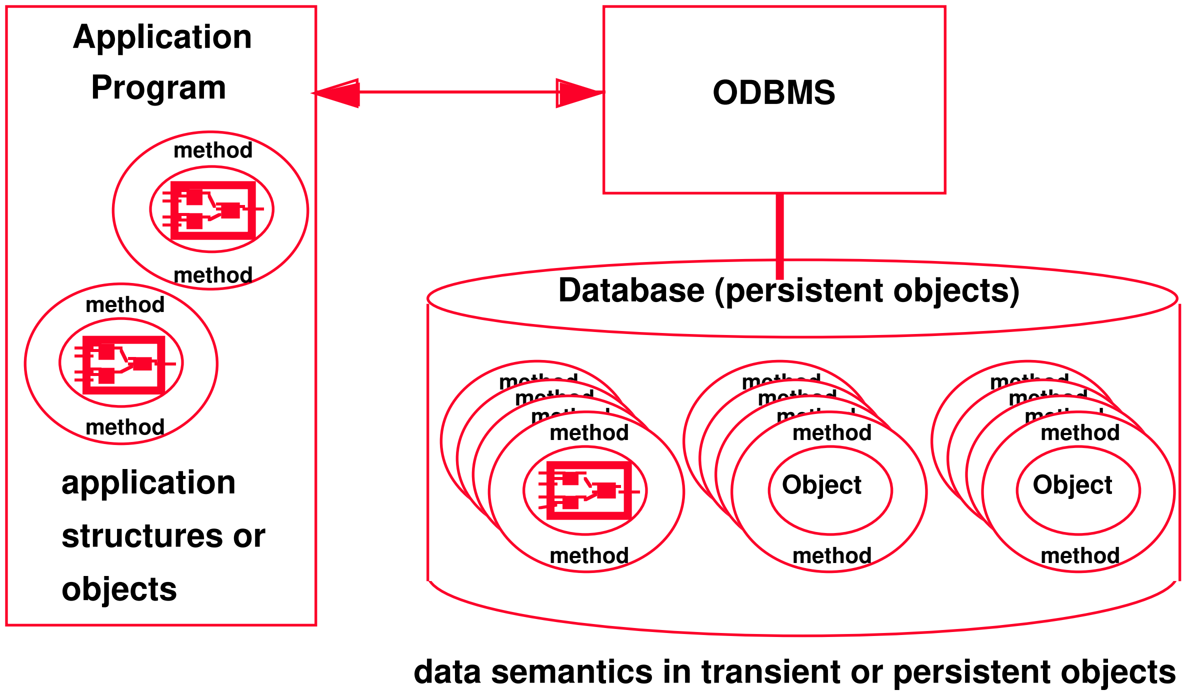

Figure 6.3.1 An Object DBMS.

In many ODBMSs, including ONTOS, ObjectStore, Objectivity/DB, and VERSANT, objects are defined as classes in object programming languages (e.g., C++) and the definitions registered as persistent types in the ODBMS. The procedures (methods) associated with these classes are compiled with the applications in the usual way, and are stored in the resulting programming language binary files, rather than in the database. When the application is executed, the methods executed are those compiled in the application. The object database itself contains only object states, in the form of persistent data structures, which are used to materialize complete objects in the application program's memory when these objects are accessed by the application. However, this fact is *not* visible at the application program interface. Instead, the application behaves as if complete objects were actually stored in the database.

This form of ODBMS, which is not itself capable of executing object methods, and requires that an object be moved to (actually, materialized in) the application workspace before one of its methods can be executed, is sometimes referred to as a *passive* ODBMS. Such ODBMSs generally do not separately execute object methods because their goal is primarily to provide a seamless persistent programming environment for the programming language, rather than providing the appearance of two separate execution environments (the database and the application program). Figure 6.3.2 illustrates a passive ODBMS.

Because object definitions in passive ODBMSs are contained both in application program files and the database, it is necessary to take care that these definitions remain synchronized, since typically it is possible to modify them independently. For example, when opening a new database or when operating on a database with a modified schema, the ODBMS must ensure that the application program's definitions are consistent with those in the database [Cat94b].

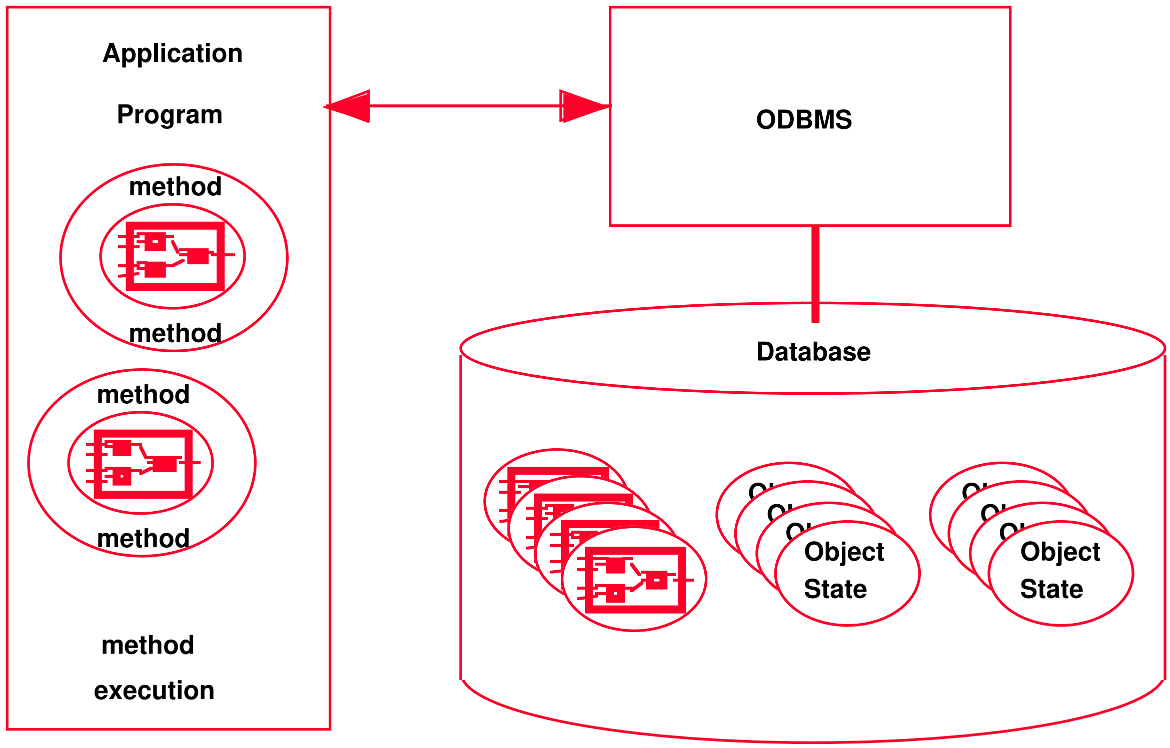

Figure 6.3.2 "Passive" ODBMS

On the other hand, in what is sometimes called an *active* ODBMS, the ODBMS actually stores complete objects, including their methods, and these methods can execute in a separate ODBMS execution environment. GemStone, O_2, ITASCA and the ORDBMSs (Illustra, OpenODB, and UniSQL/X) are examples of active ODBMSs in this sense[17]. For example, GemStone provides as part of its ODBMS environment a complete implementation of Smalltalk (Smalltalk DB), which can be used to implement complete objects (including methods) in the database. Applications can explicitly send messages to these GemStone objects to have them execute these methods. Similarly, OpenODB provides facilities for executing methods written in either OSQL (which includes programming capabilities) or other languages (external functions), directly within the ODBMS. Figure 6.3.3 illustrates an active ODBMS.

In an interface to an object-oriented programming language, an active ODBMS potentially presents the appearance of two execution environments to an application, because objects can execute either in the application or in the database. However, an active ODBMS need not present the appearance of two execution environments. If desired, an active ODBMS can provide a seamless interface, similar to those described for the passive ODBMSs, in which the existence of separate database objects is concealed from the application. In this case, the database methods could be used in situations where it is especially desirable for methods to reside or execute in the ODBMS, with application methods being used in other cases. Applications could appear to invoke methods in programming language objects, but in some cases these invocations would transparently be sent to corresponding methods in

---

[17] This use of the term *active* is not the same as the use of *active* in the research literature to refer to a DBMS that is capable of monitoring events and triggering the execution of procedures or transactions in response to event occurrences [DBM88, HLM88, BOHG+92]. The Illustra alerter and rules facility mentioned in Section 4.2 makes Illustra active in this latter sense.

the database for execution there. GemStone and O_2 are examples of active ODBMSs that attempt to provide seamlessness between application and database execution environments, at least for some of their programming interfaces. Without such a seamless interface, the distinction between objects in the application and objects in the database is more apparent to the programmer.

Active ODBMSs allow easier centralization of the definition of object behavior. This is particularly useful for certain key types of behavior, such as business rules and integrity constraints, to ensure that these semantics are maintained and enforced consistently on all applications. This type of centralization can also be done with a passive ODBMS, but often requires additional software (e.g., repository facilities). Active ODBMSs also allow some flexibility in where objects execute in the network.

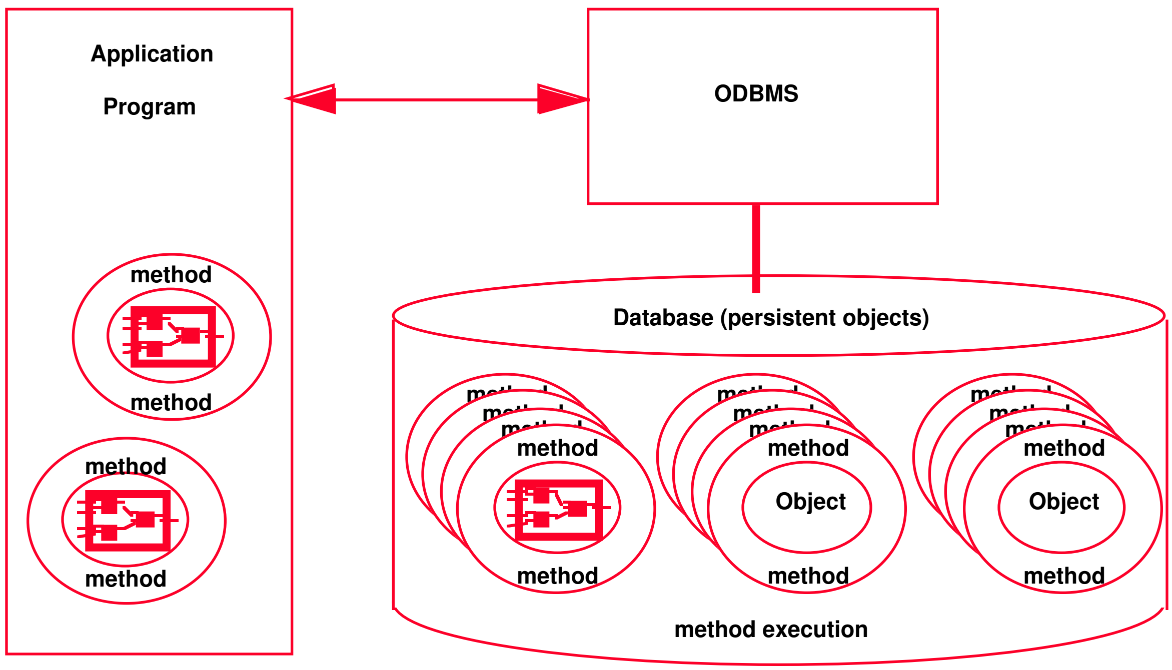

Figure 6.3.3 "Active" ODBMS

The issue of whether an ODBMS is active or passive is sometimes confused with the architectural issue of whether an ODBMS is an object server or a page server. However, these issues are actually distinct. As noted in Section 6.2, the object server / page server distinction has to do with whether the object or page (respectively) is the unit of transfer between the client and server components of the ODBMS. However, there is no requirement that the "objects" stored and transferred between components in an object server architecture actually be complete objects with methods. They may instead be the data structures used to materialize complete objects in a passive ODBMS. Similarly, there is no requirement that either component in an object server architecture actually be capable of independently executing object methods; they may simply be the components that help materialize application objects whose methods reside in the application's executable code. Correspondingly, the active / passive distinction has to do with whether complete objects (with methods) can be stored and executed in the ODBMS or not. Designating a particular ODBMS as active does not dictate whether it is the client or server component of the ODBMS (or both) that executes those methods, and thus does not dictate whether the server must deal with objects or pages. The fact that these issues are distinct is illustrated by Figure 6.3.4, which shows that ODBMS products make separate choices regarding these issues.

	object server	page server
active	ITASCA	O_2 GemStone
passive	VERSANT ONTOS (using Standard Storage Manager)	Objectivity/DB ObjectStore ONTOS (using Group Storage Manager)

Figure 6.3.4 Active/Passive and Architectural Choices for Some ODBMSs

As is the case with many of the distinctions identified between ODBMS products, it is possible to obtain at least some of the effects of one approach using the other. For example, Figure 6.3.5 shows an approach for obtaining some of the effects of an active ODBMS using a passive one. In this case, the passive ODBMS is used to implement a special application whose role is to implement the objects whose behaviors are to be subject to centralized control. This "shared client" is then accessed (e.g., using RPC facilities) by conventional applications. The combination of the passive ODBMS and the shared client provide at least some of the facilities of an active ODBMS. Of course, this may not be sufficient for some applications. (In fact, a variant of this combination is more-or-less how the active ODBMSs themselves are implemented, since object methods are not actually executed on the disk, even in an active ODBMS. However, in the active ODBMSs, the "shared client" consists of most of the ODBMS client, and all the database objects are located in this client.)

Ultimately, it may be that generalized ODBMSs will always provide some form of built-in object execution facilities. Until this happens (if it ever does), users will have to be aware of the active/passive distinction, as it may be a consideration in ODBMS selection.

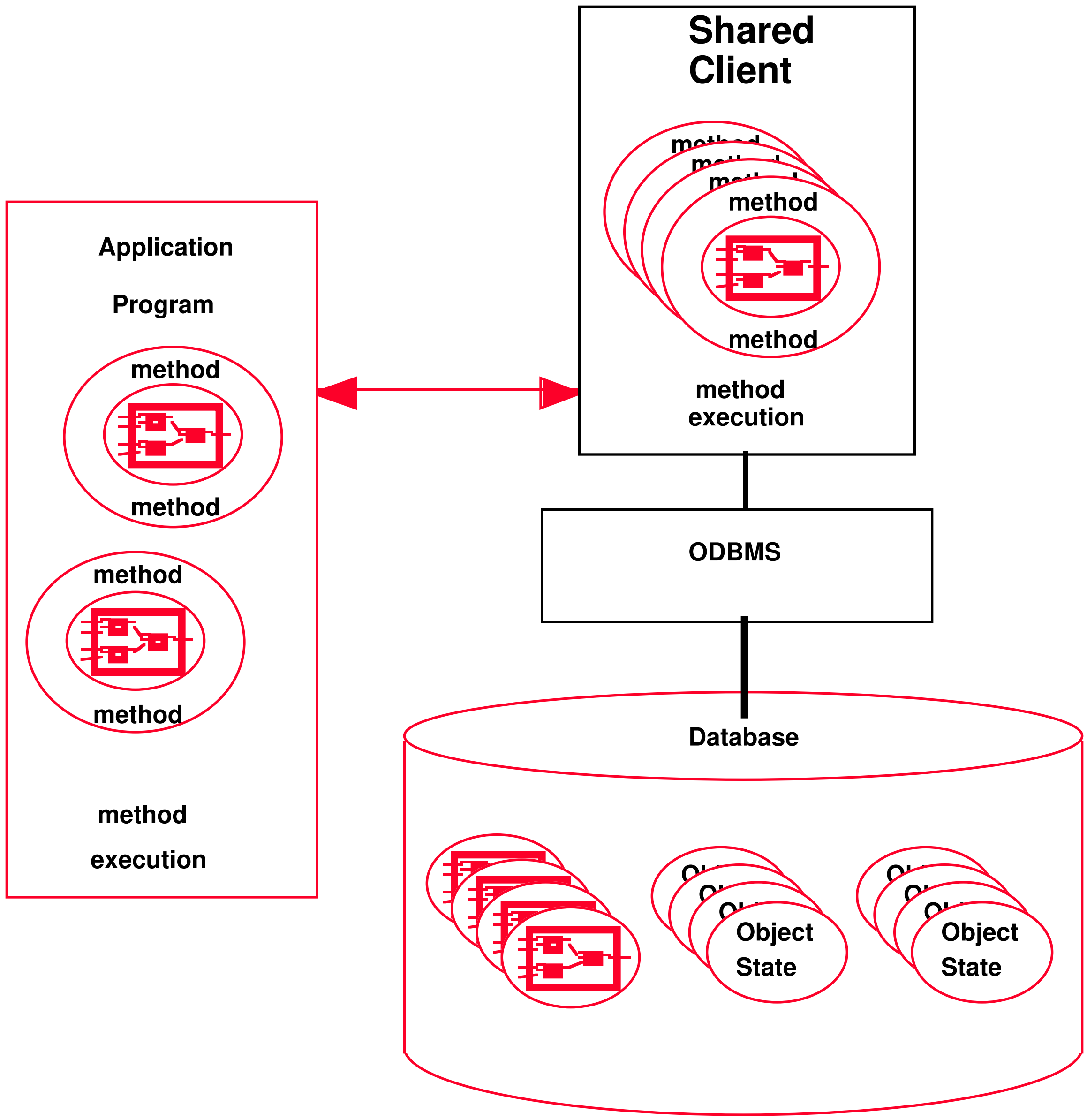

Figure 6.3.5 Passive ODBMS With Shared Client

6.3.2 Multiple Language Access

Conventionally, one of the features expected in a relational DBMS is the ability to access the database from multiple programming languages. One would like to expect the same capability in ODBMSs as well. This has proven more difficult for some ODBMSs than for others. Multiple language access in relational DBMSs is made easier by the fact that relational DBMSs generally do not attempt to provide tightly-coupled seamless interfaces with any programming language. Instead, the relational DBMS simply implements SQL. Applications are restricted to sending SQL queries to the DBMS for execution, and the language bindings involved must only deal with the relatively simple mappings required to translate atomic SQL data types, and rows, to programming language types and structures.

For an ODBMS to provide access from multiple object programming languages, the problem can be somewhat more complex, as suggested by Figure 6.3.6. The figure illustrates the concept of both complete Smalltalk objects and complete C++ objects being stored in the database. But if that is so, which object model does the ODBMS itself support, C++, Smalltalk (since there are significant differences between the object models of these languages), or some third alternative? Moreover, how could such an ODBMS be adapted to support objects from yet another object programming language (e.g., CLOS) at some later time?

The complexity of supporting multiple language interfaces in an ODBMS depends on how seamless an interface to the various languages the ODBMS attempts to provide. If the interface is not particularly seamless, the problem can be approached much like the problem of interfacing a given programming language with a relational DBMS. The program simply moves object state in and out of the ODBMS. If the ODBMS is active, the program can also invoke operations on objects located in the database, and have them executed by the ODBMS. Each language interface must separately address the problem of providing the ability, from within that particular language, of manipulating the structures contained in databases controlled by the ODBMS, and of moving data back and forth between the program and database.

The problem is somewhat more complicated for those ODBMSs that attempt to provide seamless interfaces with programming languages, since the objects in the database must then appear to behave like normal programming language objects. As indicated in the ODBMS descriptions in Section 3, several ODBMSs, including GemStone, VERSANT, O_2, and ObjectStore, provide such interfaces for both C++ and Smalltalk. As noted in [Cat94b], storing persistent objects from more than one programming language in a database, yet sharing data structures between the languages, requires mapping all data structures from one language into data structures in the other. Generally, the ODBMS will be better integrated with a specific primary language and allow more limited interoperability with others. Different ODBMSs exhibit different variations on this general theme. For example, GemStone is an active ODBMS that internally implements the Smalltalk object model, and so a tight coupling with Smalltalk is relatively straightforward. In this case, the C++ interface requires somewhat more work to implement. On the other hand, VERSANT is a passive ODBMS that internally implements its own generic object model, which primarily supports only object state. As a result, VERSANT must mainly deal with the issues of mapping state from its internal model to state in multiple languages. The programming language which an ODBMS most "naturally" supports can be an issue in ODBMS selection if it is anticipated that most applications will be developed in a single language.

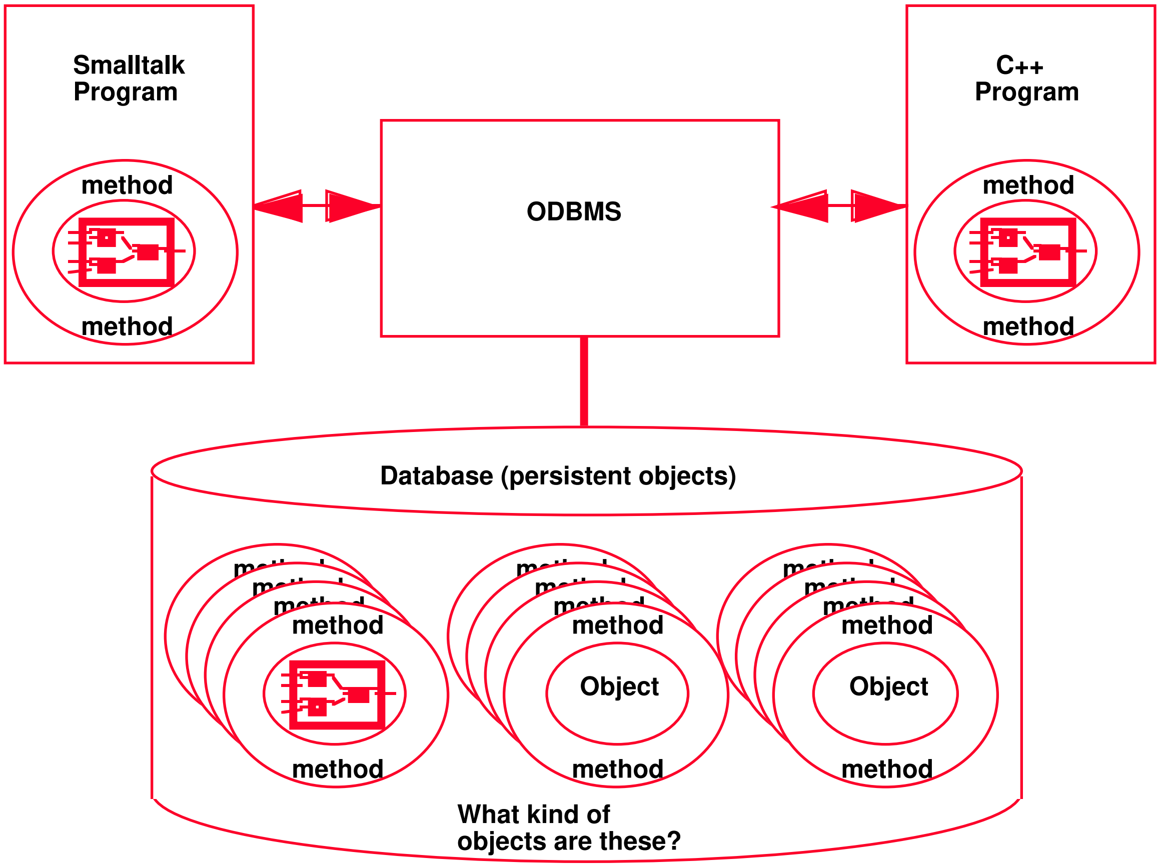

Figure 6.3.6 Multiple Language Access

6.3.3 Locating Object Behavior and Binding To It

One of the aspects of supporting object behavior (whether in an ODBMS or an object-oriented programming language) is how the code implementing particular object behavior is located and invoked when it is needed. As noted in Section 2, one of the characteristics of a typical object model is *operation overloading*. This means that there may be more than one piece of code (method) that implements a given operation, and the ODBMS or language must select the particular method to be executed when an operation is invoked on a particular object. How this is done, and how complicated the process is, can affect both ODBMS performance, and how complicated it may be to use the ODBMS.

For example, in Smalltalk, the classic object-oriented language, methods are not called directly by name, but indirectly at run time via a dispatch table associated with the class of the object to which the message was directed. If the method is not found there, the object's superclass in the inheritance hierarchy is searched, and so on upward until the root class of all objects is found. If the message is not found there, an error is reported. This allows a great deal of the flexibility generally associated with object-oriented programming, but adds additional run time overhead (e.g., method lookup and context switching versus a straight subroutine call, or even in-line code).

Dynamic message dispatching is not, however, universally used in object-oriented languages and systems. Some systems have object models that use a stronger type mechanism than Smalltalk. In those languages, it is possible to use type information to bind many, if not all, of the messages to methods at compile-time. This is the situation for C++ [Str86], and the C++ interfaces of OODBMSs. More strongly-typed versions of Smalltalk have also been investigated [JGZ88]. Recent research also indicates that the application of advanced compiler and other implementation technology to object-oriented languages and systems provides the potential for major performance improvements, even in systems with objects models *more* flexible than Smalltalk. Such techniques include compile-time binding, and caching of method addresses in frequently-used objects (so that after the first message dispatch, subsequent messages of the same type effectively become subroutine calls). Examples of such research include [CUL89, CDMB89, DMV89]. Such techniques are now frequently used in implementations of Smalltalk.

Both Smalltalk and C++ are *classical object models*, in which methods are associated with a single class. This restricts the search for a method to handle a particular operation request (e.g., in the Smalltalk example above, the search is restricted to the receiving object's class and its superclasses). Most ODBMSs use a classical model, since the major object-oriented programming languages are of this type. On the other hand, as noted in Section 2.10, a *generalized object model* does not distinguish a message recipient from other request parameters. Such models (used, e.g., in OpenODB and the SQL3 specification) allow additional flexibility in how methods are defined, and can therefore create additional complexity in how they are selected. This is because in a generalized model *all* arguments of the method are used in method selection, not just one of them. For example, the operation `display(object, device)` might have many different implementations, each specialized for a particular combination of `object` and `device` subtype. On a given invocation of this operation, the ODBMS would have to compare the combination of types used as arguments in the invocation with the combinations of types for which methods actually existed in the database, in order to choose the appropriate method. For example, suppose type `object` has subtypes `image` and `drawing`, and type `device` has subtypes `printer` and `screen`. Then the possible methods are:

```
display(printer, image)
display(screen, image)
display(printer, drawing)
display(screen, drawing)
```

In generalized object models, the search can be restricted by combining all methods for the same operation (e.g., the four methods identified above) into a single structure (called a *generic function* in CLOS). However, the rules governing which method to select for a given invocation can still be somewhat complex, since there may not be a method for each distinct combination of subtypes defined, and the rules would then have to determine which of the existing methods, if any, to invoke.

Situations requiring distinct methods for each combination of argument types only arise in cases where there is necessary polymorphism on more than one argument of the operation, as in the `display` example above. Generally, most cases only involve simple polymorphism, which the classical model can handle directly. On the other hand, if there are many operations of the `display` type, the generalized model can simplify the definition of these operations. Although the multiple polymorphism of the generalized model can be simulated in the classical model, this involves complex forms of multi-way dispatching [Ing86]. Alternatively, the object classes (and their methods) in a classical model can often be designed in ways that make the methods themselves more generic (capable of handling arguments from more than one subtype).

6.4 Support for Efficient Query Processing

Many mainstream business applications require efficient query processing (associative access to elements of large collections), as provided by relational DBMS technology. Efficient query processing is increasingly an issue (although in some respects a less-critical one) for OODBMSs as well, as evidenced by the appearance of query facilities in these products. Efficient query processing in ODBMSs raises a number of technical issues because of the generality of objects that can be stored in them. This is true whether the system is an OODBMS or an ORDBMS.

6.4.1 Form of Query Support

ODBMSs differ in the forms of query support they provide, as illustrated by the differences in the query facilities described in Sections 3, 4, and 5. In general, these different forms are based on the fact that the ODBMSs support different object models (type systems). As noted already, the OODBMSs do not currently conform to a single standard (although the ODMG standard will presumably change this), and hence can differ rather widely in the type system supported (often this is the type system of some particular programming language with which the particular ODBMS is tightly coupled). Even though the ORDBMSs generally attempt to provide SQL-like extensions, there is currently no standard for these either (again, the SQL3 standard will presumably change this).

For example, the various systems differ in *what things are queried*. In some cases, the ODBMS supports class extents (the set of all instances of a class) as the collections that are queried (e.g., ONTOS, Versant, OpenODB, and UniSQL/X). In other cases, class extents are not supported, and explicitly-defined collection objects (which must be defined by the user) are the only things that can be queried (e.g., O_2, ObjectStore, and GemStone, and tables in Illustra). In some cases, both types of structures can exist and be queried. Some systems allow queries to range over both persistent and temporary objects (this is a particular issue in those systems tightly coupled to a programming language), while in others only persistent objects can be queried.

ODBMSs also differ in *how the result of a query is defined*. In a relational DBMS, the result of a query is technically defined as another relation (table), and the particular form of relation resulting from any relational operator is formally defined as part of the definition of the relational model. This *closure* property ensures that query results have well-defined structures that can be subsequently operated on by the query language. A similar closure property has not yet been defined in a completely satisfactory way for queries involving objects in ODBMSs, although there is research work on the subject (e.g., [AK89, BK86, CDLR89, MD89]), and the ODMG OQL language (see Section 5.1) is considerably advanced in the direction of providing closure. The lack of a complete definition for closure creates difficulties in defining the query facilities of both the OODBMSs and the ORDBMSs (since the ORDBMSs also need to define the precise form of query results involving objects). In general, the result of a query might be:

- a set of existing objects
- a set of new objects of existing classes
- a set of new object states (without identifiers)
- a set of new objects of new classes

In the latter case, an additional issue is where the new classes are located in the inheritance hierarchy defined in the database schema [KKS92}. For example, ObjectStore's query facility returns sets of existing objects. ONTOS and VERSANT return values or tuples of values. These can be handled within the OODBMS type system, but are not really instances of user-defined object types (they are instances of special system-defined object types). O_2 has investigated the issue of defining query results in more depth, and thus addresses the issue of how to return new objects of new classes, and incorporate those classes into the inheritance hierarchy, with somewhat more generality. However, the object query languages in current ODBMS products frequently exhibit limitations (sometimes based on the type system they support) on the generality with which new objects can be constructed in queries, and additional work remains to be done in this area.

ODBMSs also differ in *how the query language is integrated with any programming language interfaces that may be defined*. Some ODBMSs support a query syntax that is tightly integrated with the syntax of a particular programming language. For example, ObjectStore's query facility is tightly integrated with C++, while GemStone's query facility is tightly integrated with their variant of Smalltalk. Other ODBMSs support an SQL variant as their query language, and allow query expressions in this language to be embedded within programming language statements in various ways (as in ONTOS, VERSANT, and Objectivity/DB). A third approach is to extend SQL with full programming capabilities so that it can be used directly as a programming language (as in OpenODB and SQL3).

These different forms of query support affect not only how the query facility appears to the user, but also the requirements for internal mechanisms to support the query processing. For example, the simpler the query facility, the less complicated query optimization may have to be. These internal mechanisms are briefly discussed in the next two sections. Aspects of query support (or at least of its *performance*) are also affected by the ODBMS architecture. This was noted in Section 6.2.

6.4.2 Indexing

Efficient processing of queries generally depends on the use of pre-computed results, usually in the form of one or more *indexes*. This is because searching a long collection by a sequential scan generally gives unacceptable performance with disk-based objects. For example, in a set of people indexed by name, the name index is a data structure that provides for rapid search on name, and returns a reference to one person (or more, if names are not unique).

In order to be useful for the purpose of optimizing search, an index must be an accurate representation of the collection it is associated with. This means it must satisfy a number of requirements that create potential difficulties in ODBMSs [18].

First, the index must completely represent the collection to which it refers. For example, if an index is to correctly represent a collection P of persons on the amount of their salary, the system needs assurances that every element of P has a salary (or must know what to do for a person that does not have a salary). In an ODBMS that supports inheritance, some of these persons may be employees, having salaries, and others may be dependents, not having (recorded) salaries. In this case, the need to create indexes may create a corresponding need for declaring types or other constraints on collections that insure the necessary regularity often assumed when an index is created.

---

[18] Some of the following discussion comes from [Ore89].

Second, in order to be accurate, the index must be kept up to date. For example, if a set of people is indexed by name and a new person is added to the set of people, a new entry is needed in the index. If a person's name changes, this must also be reflected in the index. The need to keep an index up to date makes it difficult to index on the output from a method or function in an ODBMS. For example, if the set of people is also indexed by the result of the `income` message, where the result of `income` is computed from various pieces of information associated with each person, it may be difficult to predict when the output from the `income` computation changes as a result of changes to other information about a person. For example (depending on the definition of the method), the result of `income` may be sensitive to changes in interest rates given by banks where the person has an account. In order for the index to be accurate, it would have to be updated in response to each such change.

Another difficulty is due to computations whose value changes rapidly. Even if the index could, in principle, be kept up to date, it would not be feasible to do so. For example, an index on `age`, where `age` is computed from a stored birthdate and a time-of-day clock, would have to be constantly updated[19].

If all dependencies can be located, for example, by analyzing the source code of the `income` method, and of methods it calls, then index updating would be possible, but this is still a research problem in the general case. Also, this approach does not work if the method value depends on random numbers, input, or anything not stored in the database. (These problems with index maintenance are, in fact, problems with maintaining the consistency of derived data of *any* kind. An index is just one possible form of derived data that may be maintained by a DBMS.)

Indexes could also be based on the internal structure of objects, rather than their responses to messages. Such indexes on stored values, which may be references to other objects, are much easier to update since the relevant updates are fairly obvious. However, indexing on structure violates the privacy of objects, since it bypasses an object's encapsulation boundary. It also requires definition of constraints on the object, as illustrated by the constraints that must be declared in the GemStone example of Section 3.1.1. One suggestion for dealing with the privacy violation issue has been to reveal parts of the internal object structure in the object protocol, and only allow indexes on revealed structures [GM87]. An alternative in ODBMSs supporting the concept of attributes or properties visible at object interfaces would be to only define indexes for those attributes. However, if an attribute was actually computed, this would involve indexing on the result of a method execution, as before.

If the index is to support range queries (for example, on salary values), the system also needs a declaration that all salary values will be comparable according to some total order, and that the behavior of the corresponding operators is "reasonable". For example, =, <, and > can be defined for rational numbers, and an index keyed by instances of a rational number class can be created. As long as the optimizer knows (because the class definer asserts it) that the functions implementing =, <, and > for rational numbers behave "as expected", optimization can proceed. The Illustra ORDBMS described in Section 4.2 provides for declaring this type of information.

---

[19] Of course, the index could always be updated only when the index value is actually needed (i.e., when a query is being processed), in which case an up-to-date value would always be available. However, this approach would not make the index very useful in improving query performance.

Finally, ODBMSs introduce additional structuring concepts that can complicate indexing. For example, in many ODBMSs, there is no concept of an automatically-maintained extent for an object class (the set of all instances of the class). Instead, there are user-defined sets of objects (tables containing ADT instances, in the case of SQL3). Any indexing would have to be performed separately for these sets, and such indexes would have to be separately maintained as objects were inserted or removed from these sets.

6.4.3 Query Optimization

Efficient query processing depends not only on the ability to define indexes, as described above, but also on query optimization that can make use of such auxiliary structures in improving the performance of high-level descriptions of the required data or objects. Much query optimization research has been done on the problem of optimizing conventional relational queries. Current research is focusing on extensions to these techniques to support queries in ODBMSs.

In the relational data model, there is a fixed set of operations (the *relational algebra*, e.g., selection, projection, join) over relatively simple structures (normalized relations). These structures, in turn, have a fixed set of access methods defined for them in the DBMS's implementation. The formal properties of the relational algebra define when an *equivalence-preserving transformation* exists for a sequence of relational operations; that is, when the sequence can be translated into a different sequence of operations that produces the same result. Cost information associated with the access methods supporting the various operations can then be used by the query optimizer to determine when the new sequence can be performed at less cost than the original sequence.

ODBMSs, however, are extensible: arbitrary data types and associated operations can be defined and added to the system, together with access methods to support them. Queries in an ODBMS can contain arbitrary combinations of such user-defined operations. Each new type, by introducing new operations, creates a new algebra whose properties are unknown to the query optimizer. Without knowledge of these new operators, or their algebraic (equivalence) properties, the query optimizer cannot discover equivalence-preserving transformations. Moreover, in an ODBMS the storage and other implementation details (such as associated access methods) of the objects are generally encapsulated within the object's interface (and in any event are not known to the ODBMS or its query optimizer ahead of time). The query optimizer cannot determine when a transformation is less expensive than the original when cost-relevant information about the implementation is hidden in this way. Also, even when information such as method code is visible, it may be difficult to determine relevant cost information based on an analysis of the code alone.

To see how these considerations affect optimization, suppose the relations

```
Part(part_number, weight, made_at)
Plant(plant_id, country, numb_employees)
```

where `made_at` contains the `plant_id` at which the part is made, are queried as follows:

```
SELECT part_number
FROM Part P, Plant PL
WHERE P.made_at = PL.plant_id
AND P.weight > 50
```

```
AND PL.country = 'USA';
```

The ordinary heuristics used in relational optimization would cause the optimizer to try to perform the selections `P.weight > 50` and `PL.country = 'USA'` prior to performing the join indicated by `P.made_at = PL.plant_id`, because selections are generally less expensive than joins, and can also be expected to reduce the number of rows to be joined. However, in an ODBMS some of the parts might be `3Dsolid` parts, as described in Section 2.1, and their weights might have to be determined by a complex computation involving the parts' density and (solid) geometry. In this case, it might make sense to perform the join before performing the selection `P.weight > 50` (if the join is very selective and weight is expensive to evaluate).

In addition to these considerations, pointer traversal (pointer chasing) generally plays a major role in ODBMS operations, and this must be considered in performing optimization in addition to relational joins (although in some cases, pointer traversals can be considered as another *kind* of join). ODBMSs also use more complex indexing techniques, as discussed above. Moreover, the clustering techniques that play an important role in the use of the OODBMSs can also have an important effect on query cost computations. As a result, ODBMS optimizers must be both more complex, and extensible. Extensible, description-driven query optimization is still a very active research problem. Examples of recent work in extensible query optimization for ODBMSs include [GCDM+93, HFLP89, MDZ93, MZD92, PHH92, SS90].

Information for use in ODBMS query optimization can be provided from a number of sources. As noted above, one type of information that is important in query optimization is information about the operations being performed. These operations include the methods defined for individual user-defined object classes, as well as the built-in operations that may be defined for system-defined collection object classes referred to in the ODBMS query language.

Information about object methods could theoretically be determined by analyzing the arbitrary programming language expressions that define object methods to determine optimization opportunities. However, ODBMSs today generally restrict optimization to a specialized query language. This is either a SQL variant, or a specialized query syntax added to a programming language (C++ extensions in ObjectStore, Smalltalk extensions in GemStone). One of the motivations behind the SQL3 extensions supporting full programming capability is the hope that the resulting language can be used to write object methods that will be more amenable to optimization.

Instead of actually analyzing method code, various *properties* of methods (e.g., their relative costs, the sizes of temporary results relative to the sizes of the operands, algebraic properties of the operations) could be explicitly specified when defining methods. These descriptions could then used by an optimizer in determining the optimal strategy for processing a given query that involves these application-defined operations. The *modifiers* that are included in object function definitions in the Illustra ORDBMS (see Section 4.2) provide examples of this type of information.

In addition, built-in operations on system-defined collection objects could be defined that form an *object algebra*, corresponding to an extension of the relational algebra described earlier. These built-in operations would be used in defining the semantics of object query languages, could be used to provide extended object manipulation facilities whose properties would be known to the optimizer, and thus could be the basis for optimized object queries. Object algebras have been mentioned earlier in this report. An example of recent work in this area is [LMSV+93].

In addition, an optimizer can make use of information about indexes that may be defined, clustering, and other aspects of the physical structure of the database.

Optimizer technology is beginning to be found in OODBMS products. ObjectStore, O_2, and GemStone are among the OODBMS products having query optimizers, while all the ORDBMS products mentioned have them. These optimizers have, however, different capabilities. For the most part, the query optimizers in OODBMSs do not do the complex, cost-based type of optimization that relational DBMS query optimizers do, although they will take advantages of indexes if they are available; the ORDBMS products, based on the more mature relational optimization technology, generally do better here. On the other hand, the ORDBMSs often do not do as good a job at optimizing pointer traversals as the OODBMS products, since in this case the OODBMS technology is more mature. Performing good optimization of queries that incorporate arbitrary mixtures of pointer traversal and other operations remains a research goal.

6.5 Support for Shared Access, Transactions, and Recovery

OODBMSs, and some ORDBMSs, generally provide mechanisms for supporting shared access and transactions that extend beyond the corresponding capabilities found in conventional relational DBMSs. Transaction support is used to deal with problems arising from concurrent access to shared data. The classic problems of "lost updates" and "inconsistent retrievals" result from not having such support [BG81]. ODBMSs, like relational DBMSs, support conventional "short" transactions. These transactions are atomic units of computation. The system may interleave the execution of several transactions (to improve resource utilization and throughput), but it guarantees that these transactions satisfy the *serializability* criterion, i.e., that the effect of a group of interleaved transactions is the same as if those same transactions had been performed in some serial order. Serializability assumes that transactions do not know about each other, and do not want to deal with concurrency explicitly themselves.

For the advanced applications of interest to ODBMSs, this type of transaction is sometimes too restrictive. This is so for two reasons. First, some of these applications deal with long-lived activities (e.g., design, software development and maintenance, problem solving, document creation) and large, complex objects. A long-lived transaction of this type by one user will either lock a large part of the database for a long period (the *pessimistic* approach), preventing it from being used by others, or will defer setting locks but will run the risk of aborting when it tries to commit (the *optimistic* approach). In such applications, the amount of work that must be undone after such an abort may be unacceptable. Second, the conventional model assumes that concurrent transactions necessarily *conflict* and hence must be precluded from concurrently accessing shared data. However, objects in an object-oriented system may span the continuum between objects that correspond to values of variables in traditional programming languages and the large, complex objects mentioned already. Contention is unlikely on objects that correspond to program variables. Imposing pessimistic concurrency control on objects of this type is an unnecessary overhead. Moreover, for some advanced applications, a different paradigm is appropriate, namely that the concurrent programs are *cooperating* to perform some global task (design, planning, scheduling, problem solving, etc.). In this case, it may be desirable for some programs to "see" the partial results of computations performed by others. As a result, ODBMSs frequently provide support for additional types of transactions, based on these special requirements. These include "long" and nested transactions, and so-called "design" transactions which may give up atomicity or serializability, or may use versions[20].

Sometimes comments are heard to the effect that OODBMSs do not provide adequate concurrency control. These are usually based on these *additional* requirements, and do not mean that the OODBMSs do not provide the forms of conventional concurrency control found in relational DBMSs. Moreover, to the extent that ORDBMSs attempt to include additional concurrency control mechanisms to handle the new applications, the same technical problems of integrating these mechanisms with existing ones will have to be solved for these extended relational systems as well.

Concurrency control in conventional relational DBMSs is typically implemented by locking (sometimes referred to as *pessimistic* concurrency control, because this mechanism assumes that transactions will conflict). ODBMSs generally also support locking, although this can sometimes be different than locking in a relational DBMS due to the existence of potential locking units that are not present in relational DBMSs (classes, class hierarchies, objects). The advantage of using locking is that transaction commits always succeed (since they will have locked the resources they need in advance of using them). The disadvantage is reduced concurrency, since concurrent applications must wait to access objects locked by another transaction.

Some ODBMSs (e.g., ONTOS and GemStone) also support an alternative called *optimistic* concurrency control, in which transactions operate as if they will not conflict. When a transaction attempts to commit, the system checks to see if it conflicted with any other transaction and, if it did, it is aborted. The advantages of this approach are that transactions that only read never delay or fail, and performance is improved in situations where there is little conflict. The disadvantage is that a long transaction may fail to commit due to a conflict, and the work done by the transaction may be lost. Where the option is provided, pessimistic concurrency control can be used for such transactions to ensure they will ultimately commit.

ODBMSs also sometimes provide various extensions to locking. For example VERSANT allows non-two-phase locking by supporting explicit locking and unlocking, and provides numerous lock variations. Objectivity/DB supports a variation of transactions known as "multiple readers/one writer"[21], in which several transactions can read a prior consistent version of the database while a single writer is updating it (and thus creating a later version). ONTOS allows the user to specify, for a transaction, whether or not readers and writers conflict. Several ODBMSs provide facilities to communicate with applications when there is a conflict. For example, ONTOS allows a user to specify, for a transaction, whether the transaction is to block waiting for a lock when there is a conflict with another transaction, or to notify the requesting transaction immediately when a conflict is detected (VERSANT supports this option for persistent locks, which span sessions and transactions). ONTOS also supports *notification locks*. An application can read a collection of objects and set notification locks on them, allowing the objects to be changed by concurrent applications, but also allowing the original application to determine, at some later time, which specific objects were changed (e.g., so they can be re-read). Notification locks can also be used to implement a form of half-duplex communication between two

[20] Relational DBMSs frequently also provide support for transactions that relax serializability, but these facilities are generally less flexible than those in ODBMSs.

[21] This is sometimes referred to as *multiversion* concurrency control, although the versions referred to in this case are versions of the *database*, not versions of individual objects, as in the versioning mechanisms discussed below.

applications (using shared mailbox objects, and notification locks to indicate when new messages have arrived). VERSANT provides facilities for one application to notify another by email when a conflict exists on a persistent lock, or when a persistent lock has been "broken" by another application (*soft* locks, another VERSANT option). Persistent locks are used in long transactions (see below), so notification by email makes sense in this case. O_2 provides exceptions programmable on WAIT or DEADLOCK events.

ODBMSs also provide a number of different mechanisms for use in implementing "long" transactions. For example, ONTOS and VERSANT support a checkpoint facility, and Objectivity/DB supports a "commit and hold" operation, for saving work without committing the transaction. In addition, many ODBMSs provide facilities for checking objects in and out of private workspaces (often integrated with the version mechanisms described below). For example, in VERSANT, Objectivity, and ITASCA, a checkout operation copies objects to a private database. In VERSANT, persistent locks can be left behind when an object is checked out, and a long transaction may span user sessions. In ObjectStore, a checkout derives a new version of a configuration (see below) in a separate workspace, while other transactions work on other versions of the configuration in other workspaces.

In general, ODBMSs use the same sort of recovery techniques (various combinations of before and after image logging at the page or record level) as relational DBMSs do.

ODBMS applications often involve maintaining multiple *versions* of objects (Figure 6.5.1). Examples include historical and financial applications, and design applications (CAD/CAM, CASE, and document preparation). The design applications are characterized by the idea that different people can work on different copies of the same object (e.g., a part of a document). Each copy is treated as being a different version of the same object. Later, it may be necessary to merge the various versions created in parallel work to create a finished copy of the object. This may be a complex process, but in design work, it is generally considered better to do it this way, due to the parallelism it allows. Versioning facilities generally allow for different versions of the same object, and also the designation of one of the versions as the "default" version (e.g., the most recent version). Versioning may permit new versions to be created only by modifying the latest version (*linear* versioning) or may permit new versions to be created by returning to an earlier version and creating *branching* versions. Versioning facilities also must provide for maintenance of relationships among versioned objects (e.g., different versions of object A may need to refer to different versions of object B, but these versions may not be created in a synchronized way). ODBMSs providing version support include ObjectStore, Objectivity/DB, VERSANT, ONTOS, ITASCA, and GemStone. Typical operations on versions include operations to create and delete versions, create and delete version branches, merge versions, and get specific or default versions.

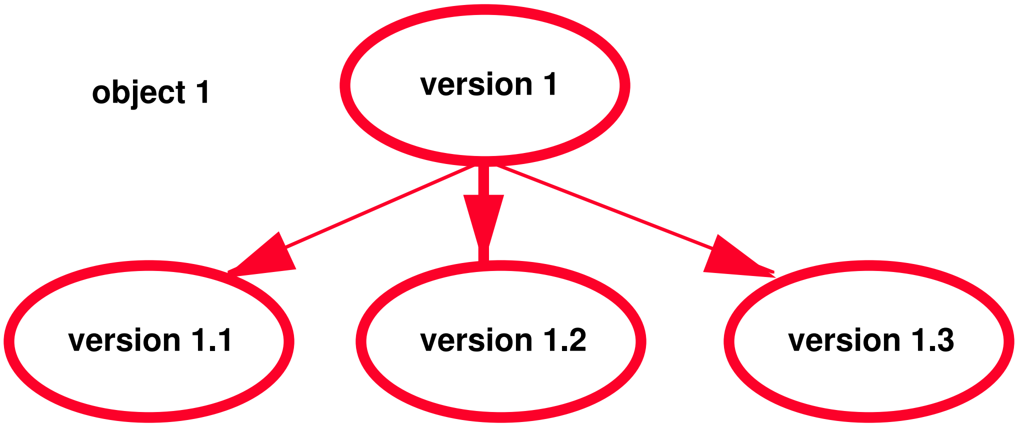

Figure 6.5.1 Object Versions

Some ODBMSs provide support for object *configurations* in addition to providing version support. A configuration is a group of objects that is to be treated as a unit for versioning purposes. For example, in Figure 6.5.2, each configuration consists of one version of objects A, B, C, and D, with the version of A referring to the versions of B and C as shown. Configuration 1.1 is derived from configuration 1 by replacing version 1 of objects A and D with version 2 of each of those objects. Configuration 1.2 is derived from configuration 1 by replacing version 1 of object B with version 2. Objectivity/DB, ObjectStore, and ONTOS provide examples of configuration facilities in ODBMSs.

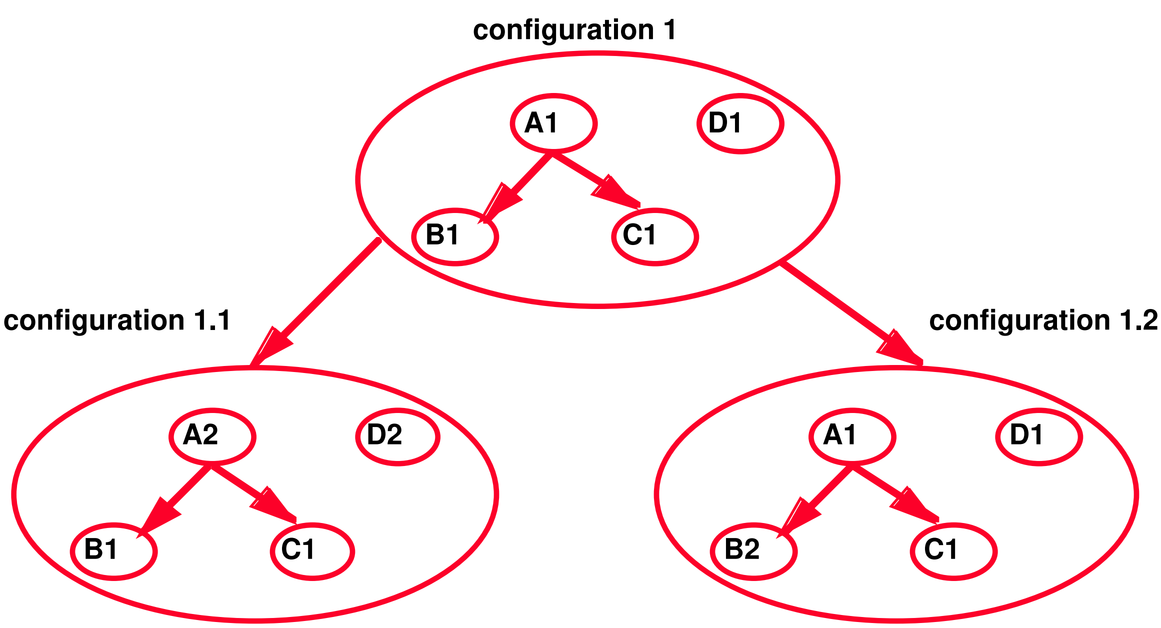

Figure 6.5.2 Object Configurations

Objectivity/DB takes the approach of providing basic facilities to assist users in defining configuration management policies. In Objectivity/DB, inverse attributes representing relationships in versioned objects can be tagged with a *move*, *drop*, or *copy* option. This specifies the action to be taken when one object points to others, and a new version of that object is created. If *move* is specified, the reference is moved to the new version and set to NIL in the old; if *drop* is specified, the reference stays with the old version, and is set to

NIL in the new; if *copy* is specified, both the old and new versions have the same reference. For example, in Figure 6.5.3, version 1 of object A points to objects B and C, and these have inverse relationships tagged with the *move* option pointing back to A. When version 2 of object A is created, the move option specifies that the references to A in objects B and C are adjusted to point to the new version of A, and the references to B and C in the old version of A are moved to the new version, and set to NIL in the old version.

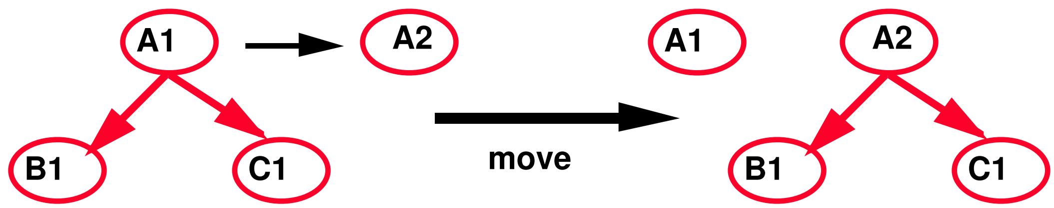

Figure 6.5.3 Objectivity/DB move Option

ObjectStore takes the approach of providing built-in configuration semantics. ObjectStore represents configurations as special objects of class `os_configuration`. Objects are specified as being in a particular configuration at the time they are created, by passing the configuration as an argument to the persistent `new` operator. Class `os_configuration` defines various operators for manipulating configurations, such as operators for checking configurations in and out, creating or destroying segments of configurations, and freezing configurations that are no longer to be writeable. ONTOS provides a somewhat different built-in configuration concept.

The type of versioning described above does not really address the needs of many kinds of historical data [CT85], only historical data that can be organized in discrete versions. Illustra provides a feature called *time-travel* that enables a user to query data which was current at a specific point in time, or during a period between two points in time, other than the present. Points in time can currently be expressed only in terms of absolute time references. Time travel specifications are contained in the table reference of a table being queried. For example, the query

```
select fname, lname
from employees;
```

selects current data from the `employees` table. On the other hand, the following query selects data from the same table that was current on January 1, 1993:

```
select fname, lname
from employees @['Jan 1, 1993'];
```

or between January and the present:

```
select fname, lname
from employees @['Jan 1, 1993', 'now'];
```

Time travel is only available for `select` statements; it cannot be used to modify old data. Illustra supports time-travel by maintaining historical data in base tables. This means that updated or deleted rows remain in their tables, although marked as non-current. This old data can be cleared out by *vacuuming* the table (using an special `vacuum` statement added to Illustra SQL or a utility). Users can specify that certain tables will be archived. When archiving is specified, vacuuming does not remove the old data, but rather moves it from base tables to archival storage. Time-travel queries can retrieve old data either from base tables or archival storage transparently to users. Illustra also supports a versioning facility, but with the *table* as the unit of versioning rather than the individual object (i.e., users can create different versions of the same database table).

Locking granularity is sometimes raised as an issue in evaluating ODBMSs [Loo92, Bar94]. In principle, locking granularity refers to which units of data can be locked. For ODBMSs, such units include entire files or databases, as well as object clusters, or individual objects. However, as an ODBMS evaluation issue, locking granularity generally refers to the unit of data that is actually locked when an individual object is locked. The most frequent comparison is between those ODBMSs that can lock individual objects, versus those that lock the entire page on which an object is located when the object is locked.

As noted in [Bar94], in page-level locking, all the objects on a database page are locked at the same time. When objects are small, a page will contain many objects, so a single read lock on a page can allow many applications to read any of the page's objects at the same time. Since the ODBMS needs to manage only one lock per page, this reduces the overhead of locking and can provide better performance. On the other hand, if one application is reading an object and one or more other applications are trying to update different objects on that same page, page-level locking will reduce concurrency. The applications trying to update objects will have to wait for the reader to finish, and then take turns updating objects as they each obtain a write lock on the page. Object-level locking sets locks on only those objects that are in use on a page, not the entire page itself. This involves a higher locking overhead, because the ODBMS has to maintain more locks. However, it also increases concurrency, because there is less chance of conflict when a given application locks only the objects it is actually using.

Evaluating this tradeoff requires knowledge of both application data and application access patterns. Page level locking generally performs well if the application involves either large objects, or accesses many objects on the same page at the same time (this is likely to be true if clustering works well), thus creating fewer unnecessary read/write lock conflicts. Object-level locking generally performs well if objects are small, and there are expected to be frequent read/write conflicts on the same page, but not necessarily on the same objects.

Whether a given ODBMS implements page-level locking or object-level locking is greatly influenced by its choice of architecture (see Section 6.2). However, even this choice is not entirely definitive. For example, [CFZ94] reports on several new approaches for supporting object-level locking in page server architectures which provide improved performance over currently-used techniques.

7. Application Considerations

There is general agreement that CAD, CASE, and other advanced database applications require various extended DBMS capabilities, such as the ability to define and manipulate user-defined data types, support for complex objects, etc., as described in Sections 1 and 2. Moreover, there is general agreement that the *object* concept provides the key to providing many of the required capabilities. As described already, two general approaches to incorporating the object concept in DBMSs have been identified, namely the OODBMS and the ORDBMS, and there is also increasing agreement that both these classes of systems should support similar facilities.

What is often less clear, and less subject to agreement, are the applications for which ODBMS products should be considered. This section discusses this issue from two general perspectives. First, we briefly review a number of ODBMS applications described in the recent trade press, to give a flavor of the variety of applications in which ODBMS products are being used. Second, we discuss more general aspects of potential ODBMS applications, and attempt to identify general categories of applications for which ODBMSs should be considered.

7.1 Sample ODBMS Applications

At the time of the previous report on OODBMS technology [Man89a], relatively few descriptions of production ODBMS applications were available. Today, articles describing ODBMS applications frequently appear in the trade press and elsewhere. Some applications of ODBMSs have already been cited in Sections 3 and 4, in connection with the discussions of the individual ODBMS products, and indirectly in Section 2, which used examples from multimedia applications to illustrate various ODBMS capabilities. [Cat94b] also contains illustrations of how ODBMS capabilities apply to various application classes, such as CAD, CASE, and office automation. This section identifies some additional applications that have been recently described, particularly noting the reasons given for the selection of an ODBMS for the application.

7.1.1 Telecommunications Applications

Telecommunications applications are likely to be of the greatest interest to many readers of this report. [Ols94] has described a number of telecommunications applications, and related public utility applications (all of these were originally described in [Gar94]). One such application is RAVE, a system for monitoring telecommunications switches built by Ameritech, using the ObjectStore ODBMS. The same article described another application called ATAC, for monitoring call volumes by carrier. The article cited the advantages of using ObjectStore as being better performance and flexibility due to ObjectStore's tight integration between C++ and the ODBMS. The article also noted the advantages of being able to handle much of the processing on client workstations, as opposed to concentrating processing on a database server, and the ease of adding enhancements to the application provided by the use of an object-oriented approach.

[Ols94] also described a Trouble Management System-Network Server application, developed by Bell Canada using the GemStone ODBMS. The article again cited the ability to easily add enhancements to the application provided by an object-oriented approach, as well as the scalability and ability to partition the database to run on specific CPU's provided by GemStone's flexible client/server architecture, as reasons for choosing an ODBMS for this application. Both [Ols94] and [Ber94] described the development of ATM switch software by Fujitsu Network Switching, using the VERSANT ODBMS. This software will be embedded in ATM switches used in the North Carolina Information Highway project, as part of the operations, administration, maintenance and provisioning (OAM&P) interface. It will handle performance management and help in the set up and tear down of permanent virtual circuits.

Both [Ols94] and [Kay94] describe the Automated Mapping and Facilities Management system developed by Florida Power and Light using Smalltalk and the GemStone ODBMS. The developers noted the advantages of object-oriented software in enabling them to directly model the network and the connectivity of the various network elements, and indicated that a primary reason for choosing an OODBMS was the ability to directly map application objects into database objects. [Kay94] also indicated that the application required support for "long transactions" (see Section 6.5) during design-related application processing.

[Ols94] also described the development of NetMaker XA network management application software by Make Systems, using the ObjectStore ODBMS. The article cited the better fit of objects to the application as a advantage of using an ODBMS. Other advantages cited included the flexibility in adding new object types and behavior, and the code reduction provided by reuse in an object-oriented environment.

[Ber94] describes Versant's targeting of telecommunications applications for ODBMSs, and cites as examples the use of the VERSANT ODBMS by MCI for OAM&P, and by Claircom for operations support systems. [Eng92] goes into more detail regarding the Claircom application. As described in [Eng92], this is a network management application by Hughes Network Systems for managing Claircom's air-to-ground telephone system, using the VERSANT ODBMS. The system is composed of eight applications written in C and C++. When a passenger places a phone call, a call record is created for billing, and call statistics are captured. The database has approximately 20 defined classes. Data types include configuration data of the terminal or phone equipment, fault data for events such as power or line failures, and facilities to manage radio spectrum usage (e.g., when channels are seized or relinquished). Other data types include call quality data, accounting data, records of the number of times a channel is seized, etc.

Each ground station has its own VERSANT database in which local statistics are collected. A TCP/IP network connects each ground station with a centralized (group) database at the network management control center in Kirkland, Washington, where data from the local sites is checked in. The local sites collect up to four days of data that can total up to 20MB each. As many as 6500 objects per ground station may be involved each day. The size of the group database is expected to exceed 600MB once all ground stations are operational. Ten real-time process types may use the database for authorizing air terminals, collecting alarm events, and calling records from airplanes that fly near and use ground station equipment. The ODBMS also manages unmanned sites by monitoring smoke alarms, temperature, and other events. High reliability is a key requirement.

VERSANT's distributed database capabilities, its checkin model for delivering a group of objects to a central site, and its network performance, were cited as important factors in its selection. ODBMS performance is described as being ten times better than relational system performance, primarily due to the use of direct pointer relationships between objects. The article also cited as an advantage the flexibility in making application changes provided by an object-oriented approach.

[Ber94] also cites other possible ODBMS applications in telecommunications, including adjuncts and intelligent peripherals, ATM switch management, customer network management, head office collection and billing systems, operations support systems, remote digital terminal management, and services databases.

7.1.2 Miscellaneous Business Applications

In addition to applications specifically related to telecommunications, several articles have described various business applications that might not immediately be associated with ODBMS technology. For example, [Gar94] describes a financial application, allowing traders to buy and sell options, developed by Chemical Bank, using C++ and the ObjectStore ODBMS. The article cites better efficiency in describing financial instruments using objects as a key driver in this application.

[Kay94] describes several software products developed by Continuum, a producer of software for the insurance industry. One such product is the Continuum Workstation Platform/Enterprise Solution (CWP/ES), which helps insurers manage distributed legacy transaction systems. CWP/ES is written in Smalltalk on OS/2, and provides users operating on workstations with an interface that logically integrates data on multiple mainframe legacy systems. It uses the ONTOS DB ODBMS to maintain local (workstation) persistence of all data that users enter through this interface. A workstation server then constructs transactions using this data and sends them to the multiple legacy systems to update their databases. The existence of a local database for the workstations means that these transactions can be resubmitted later if a problem exists on a mainframe. An ODBMS was used because the workstation software (e.g., the user interface) was developed using object-oriented techniques, and it was desirable to maintain a single integrated view of data for the users (e.g., a user should be able to specify an address change once, and have it propagated to all legacy systems maintaining address information).

Another Continuum product cited is the Business Process Management/Enterprise Solution (BPM/ES), which manages work flow and resource allocation. BPM/ES was completely developed in Smalltalk, also using ONTOS DB. Resources in the system are defined as objects; similarly, tasks and higher-level work aggregates in the workflow model are also modeled as objects. An algorithm then assigns units of work to resources. The primary advantages cited for the use of an ODBMS were better integration of the DBMS with the (object-oriented) programming environment, and better modeling of the application than with a relational DBMS. This, for example, eliminates the need for complex joins in processing application requests.

Another example of a conventional business application of ODBMS technology was an object-oriented system developed for billing, cash and credit processing, meter reading, and account services applications at Brooklyn Union Gas, a gas utility in the New York Metropolitan area serving 1.2 million customers. The system was developed by Arthur Anderson [Dav89], and was described in the previous report [Man89a]. It was built in a large IBM mainframe environment (3090-MVS/ESA including CICS, TSO, and DB2) using PL/1 as its implementation language, and DB2 as its underlying DBMS. The primary reason for using an object-oriented development approach was the need for the system to support not only existing applications, but also future applications (near the mid-point of its life-cycle) involving distributed cooperative processing, and event-driven, highly interactive user interfaces. The developers anticipated that the use of the standard "message" interface in the object-oriented approach would allow an easy migration path to cooperative processing. They also believed that the object-oriented approach would allow the corporate-level definition of object kernels which could then be customized for local business units using the inheritance facilities of an object model. This would support overall control and interoperability, while retaining the flexibility needed to accommodate specialized local requirements. In order to support these requirements, the developers effectively built their own object-oriented layer over the relational DBMS, as shown in Figure 7.1. This object layer is the sort of facility provided by the Persistence product described in Section 4.4.

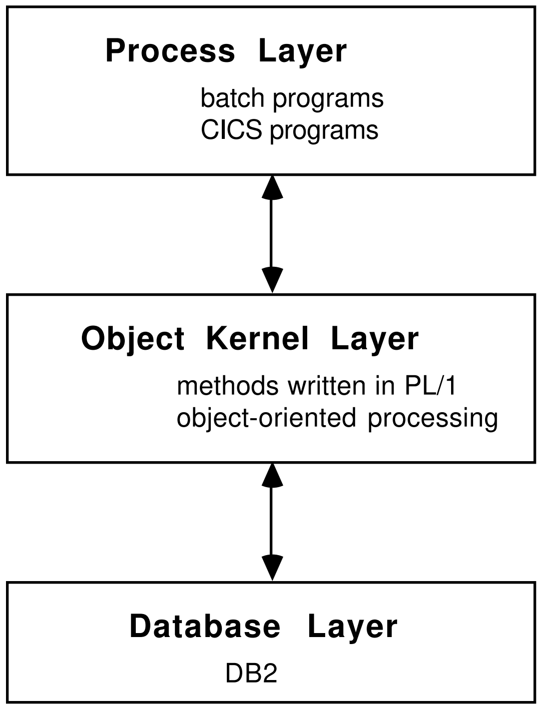

Figure 7.1 Gas Utility Application Software Architecture

[Kay94] also cites software developed by Delfin Systems, producer of InfoPower software for the intelligence community, using the ObjectStore ODBMS (this is not, strictly speaking, a "business" application, but could be considered "business-related"). The article cites the need to model relationships in a very flexible way, and to change them rapidly as new information is obtained, as advantages of an object-oriented modeling approach for intelligence applications. The article notes that exploring these relationships with an RDBMS involves doing many joins, the result being a significant drop off in performance. The ODBMS is cited as being two orders of magnitude faster, and as providing the ability to view multiple types of data images, text, etc.) in an integrated way. [Kay94] also cites the Page Systems PassagePro document management and production system, developed using the VERSANT ODBMS. The article cited the use of object-oriented programming for development, and the need for an ODBMS to match the programming environment, as a key reason for using an ODBMS.

7.1.3 Design-Related and Other Specialized Applications

CAD/CAM, and design applications in general, were the applications for which the original OODBMS products were targeted, and were the original market for those products. As a result, we do not describe a great number of these applications here. However, a few such applications are cited, in order to represent this particular class of ODBMS applications.

[Gar94] describes CAM software developed by Cimplex, using the Objectivity/DB ODBMS. The developers cite the ability of the ODBMS to model machine tools, and the objects they are processing, as key factors in choosing an ODBMS. They also note the flexibility of object-oriented development, specifically the ability to extend existing applications, and minimize coding, as important advantages of an ODBMS.

[Eng92] describes several design-oriented ODBMS applications. One of them is the CartoAssociate GIS application at National Oceanic and Atmospheric Administration (NOAA), developed using the GemStone ODBMS. The developers cited as advantages of GemStone the ability to model cartographic features using objects, the ability to more easily define aggregates (features composed of other features) and relationships between features, and better performance due to the lack of complex joins in manipulating the objects. The developers also cited the flexibility in development provided by inheritance and polymorphism in an object-oriented model. Another application cited is an electronic CAD application at AT&T Bell Laboratories Circuit Pack CAD Department, using the ONTOS DB ODBMS. The primary requirement is to model circuit boards. A separate database is created per board. The average size of one of these databases is 13-14 MB, with up to 100,000 objects being contained in a database. The primary advantage cited for an ODBMS was the ability to easily model the complex objects represented by the circuit boards. A similar application cited in [Eng92] is the DACS (Data Acquisition and Control System) real time processing monitoring and control system developed by IBM Canada's Technical Products Division using the GemStone ODBMS. DACS monitors the testing of printed circuit cards. The developers cited as the primary advantage of using an ODBMS the ability to model complex objects, noting that it would take 30-40 tables to represent the circuit cards in a relational DBMS. They also cited the ability to easily extend the application later, and better query performance due to not having to perform complex relational joins, as other advantages of an ODBMS.

Finally, a recent UseNet News posting described a realtime information platform (involving some 50 applications) used to monitor oil drilling processes, and implemented using the VERSANT ODBMS. The message noted that previous versions of the software had run on a relational DBMS (and performed well), but that the developers had encountered considerable difficulty in fitting their data to a relational model. Also, newer versions of the software were introducing retrieval time requirements that the major RDBMS vendors could not achieve. The posting noted that the current software was built to continue to support the RDBMS legacy by having applications directly access a separate Object Data Manager layer. This layer then calls either VERSANT or the relational legacy system as required.

7.2 ODBMS Application Selection

In general, ODBMSs appear attractive because of their ability to support applications involving complex and widely-varying data types, such as graphics, voice, and text, and the complex structures frequently encountered when using such types, which are not handled adequately by relational DBMSs. OODBMS applications originally focused on design functions, primarily mechanical and electrical CAD. Related applications, such as geographic information systems (GISs) and CASE were also considered for the original OODBMS products. However, ODBMSs are increasingly receiving attention in non-engineering areas, such as telecommunications, financial services and computer-aided publishing, as indicated in Section 7.1. Such applications can be as complex as engineering functions, and can also involve complex data types. For example, a page layout in a computer-aided publishing application might involve numerous different type fonts, together with pictures and graphics. An object-oriented approach enables the representation of both the characteristics of the individual layout elements (including any specialized behavior), as well as the role and location of each element in the overall layout. Such applications also share the characteristics of the design examples of requiring support for highly interactive user interfaces.

Some discussions of when an ODBMS might be appropriate for an application concentrate on citing specific application areas, such as those above (CAD, CASE, publishing, etc.). Another type of description of when to use an OODBMS (and when to use a relational DBMS) was given by Esther Dyson [Feu89]:

> The criterion is 'where is the important information?' Is it in numbers and values, or are the relationships between the things in your database what you care about? Is it the structure or is it the data? If the important information is in the data, then you can use a relational database, but if the knowledge that you care about is how things are structured and how they are related, then you probably want an OODB...If you just want an accounting of planes and ships and how much they cost, you can use a relational database. But if you want to know how parts and subassemblies relate to each other in an airplane or a ship, then you should use an OODB...The place to start is where they [OODBs] are needed the most, where structure and modeling design are the driving force.

Although these sorts of descriptions might be helpful in suggesting potential applications of ODBMSs, a more detailed analysis of application requirements is obviously necessary before drawing any firm conclusions that use of an ODBMS is indicated. For example, a description of ODBMS applications in terms of general application areas such as "design" does not really address what it is *about* these application areas that requires ODBMS technology, or what aspects of ODBMS technology are the most crucial. This would be helpful in identifying additional application areas not already thought of. In addition, most DBMS applications tend to "run together". Organizations that do extensive design or publishing have payrolls and (if they are to remain in business) orders and customer information, i.e., classes of information that have generally been handled reasonably well by relational DBMS technology. One of the issues in a context like this is where the "boundary" exists between the use of ODBMSs and the use of conventional relational DBMSs. Similar comments might be made about characterizing ODBMS applications in terms of an emphasis on "structure" or "relationships" as opposed to "data". After all, the structure is *part of* the data, and *relational* databases are supposed, among other things, to handle *relation*ships.

To a rough approximation, the assumptions generally made when using a conventional relational DBMS are one or more of the following:

1. A typical application's access to DBMS data can be reasonably characterized in terms of associative (content-based) selection, expressible in a high-level language (such as SQL), performed over large sets of data. This means that:

   a. it makes sense to organize the database data in a way that makes this type of processing efficient (normalized relations)

b. the most important task of the DBMS is the efficient, optimized processing of such high-level requests.

2. There is a fairly close match between the structures manipulated by the DBMS (the normalized relations) and those that must be manipulated in the typical application's main memory workspace. This means that, using operations on normalized relations, the DBMS can return data in a form reasonably close to that required by the application, and there is little overhead imposed on the application to reformat the data returned by the DBMS into the form which the application needs to manipulate.

3. There are numerous applications having widely-varying requirements for data in the DBMS. Based on this assumption, it makes sense to maintain data in a "least common denominator" form (normalized relations), and construct specific combinations of data to be returned as they are required.

The position taken by the developers of the original OODBMS products was generally that they were developing OODBMSs to address database requirements of engineering applications (and other advanced applications such as computer-assisted publishing) that began to depart significantly from these assumptions. Of course, the question is how significant a departure must there be from these assumptions in order to justify the use of a new class of DBMS product (this includes the ORDBMSs as well as the OODBMSs, since the ORDBMSs are new products with, in some cases, less maturity than some OODBMSs).

Clearly, other things being equal, a DBMS that allows some essential function to be performed is better than one that does not. However, usually that is not the choice that is faced, since the function can usually be performed in more than one way (or by more than one component of the system, e.g., either in an application program or in the DBMS). Thus, the choice usually involves determining which approach gives the best figure of merit over a *set* of considerations including:

1. required performance for the critical applications

2. adequate performance for the other applications

3. ease-of-use characteristics

4. software development life cycle costs (e.g., maintainability, ease-of-development, modularity and data/code sharing, extensibility to support new requirements, interoperability with current and future software)

5. price-performance ratio (e.g., for the application in question, what size hardware platform is required to enable the relational DBMS to equal the performance of the ODBMS, or vice-versa).

Determining these things clearly requires a detailed understanding of both the application requirements, and the characteristics of the various DBMSs under consideration. An analysis of this type for even a single specific application and a single ODBMS is beyond the scope of this report. However, the following sections attempt to provide some additional ideas related to ODBMS application selection by roughly categorizing applications, not by general application areas, but by the primary features the application demands of the DBMS. According to this approach, ODBMS applications generally fall into three rough categories:

1. Applications that need both object *features*, and whose performance requirements also demand specialized ODBMS *implementations*.

2. Applications that need object features, such as extended data types and other object facilities, but do not necessarily require specialized implementations to perform adequately. Such applications could, for example, perform adequately with object features added to what is basically a relational database engine.

3. Applications that, for reasons of software engineering (e.g., life-cycle cost issues), adopt an object-oriented development approach in a situation that is otherwise fairly conventional, and require a DBMS that fits into this approach.

These are application categories that *currently* suggest the use of an ODBMS. Based on current trends, as well as on the incorporation of object features into the SQL relational standard as described in Section 5.2, *all* DBMSs will ultimately have object features. As a result, DBMS selection criteria will ultimately be based on the specific details of individual products, including both supported features and architecture/implementation characteristics.

The categorization above suffers from problems similar to those of categorizing by application area, in that both types of description are rather vague. However, as additional experience becomes available on how ODBMSs actually perform in various application environments, and as better performance measuring techniques become available, this sort of high-level discussion will hopefully become less and less necessary. Other factors making such categorization less necessary will be the merger of features just described, the increasing conformance of ODBMS products to standards such as ODMG and SQL3, as well as the increasing use of ODBMS architectures that allow heterogeneous types of ODBMS to interoperate, and thus allow the use of multiple products tailored to different aspects of application requirements within the same application architecture.

7.2.1 Applications Requiring Object Features and Specialized ODBMS Implementations

The applications in this category are applications, such as the more intensive CAD applications, that tend to depend on specialized ODBMS implementations to provide the required performance. The ORDBMSs described in Section 4 show that most, if not all, of the features provided by OODBMSs can also be implemented within a relational framework. As a result, the distinction between this category of applications and the category described in Section 7.2.2 has primarily to do not with whether various externally-visible object *features* (e.g., user-defined types with behavior, inheritance) are available, but rather with how they are *implemented*.

The "specialized implementations" referred to are those that support characteristics of object processing that differ significantly from characteristics typically found in conventional relational DBMSs. These include support for seamless programming language interfaces, and for efficient fine-grained interleaving of application processing and access to persistent objects. This support generally requires the use of one of the *data-shipping architectures* [CFZ94] (see Section 6.2) found in many of the current OODBMS products. In such an architecture, data is shipped from servers to clients so that much of the ODBMS's processing (sometimes including query processing), as well as application processing, can be performed at the client. The interfaces provided by these ODBMSs generally emphasize navigation through the complex graph-like structures illustrated in Sections 1 and 2, requiring that the application follow direct references from one object to another. The data-shipping approach moves the data closer to the application, allowing an individual persistent object to be accessed quickly, and hence supporting the efficient fine-grained interleaving between application processing and access to the next persistent object required by the application.

Tools like a design rule checker in an electrical CAD application require that, given one component, the system must be able to quickly reference the other components connected to it. For example, if a program is working on a circuit board, it will often require the backplane that the board is connected to. Although this kind of access can be viewed as a degenerate query, other implementation techniques are generally more useful for this type of access than techniques that have been designed for queries over large sets. As a result, these applications typically involve the manipulation of complex graph structures (the *complex objects* mentioned in Section 1), in which objects related to a given object are identified by direct references (typically pointers) to the related objects. The key to performance in these applications is the speed with which the application can traverse these structures by following these references to related objects.

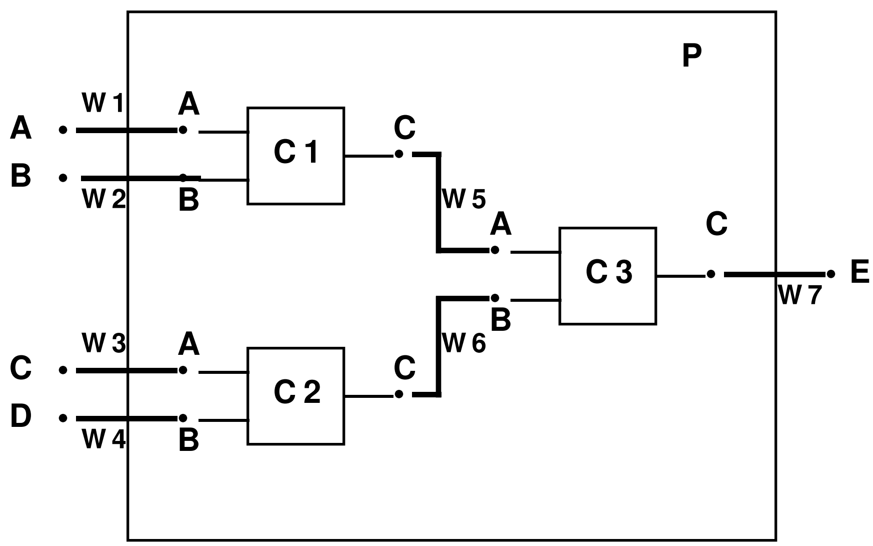

Figure 7.2 A Simple Object in an Electronic CAD Application

For example, Figure 7.2 shows an (oversimplified) object that might be used in an electronic CAD application (the 4AND example used in Section 1). The figure itself illustrates the graph-like structure of the object, in which each gate, pin, and wire must be represented. The application would spend a great deal of time traversing this structure (e.g., following a wire to the component at the other end), and thus this form of access must be as efficient as possible. Similarly, in a CASE system, each box and connection of a dataflow (or other) diagram must be represented, and the application must, for example, refresh its video display by accessing individual objects.

As noted in Section 1, in order to explicitly represent the object of Figure 5.1 in a relational DBMS, each gate and wire of the schematic would have to be represented by one or more tuples. In order to follow some connection, e.g. from a pin to a wire, a join would be required. To do any useful work, e.g. a circuit simulation, a very large number of joins would be required. Similarly, in a CASE system, each box and connection of the dataflow diagram would have to be represented by one or more tuples, and the application would have to refresh its video display by querying the database for each of these elements.

As also noted in Section 1, these relational operations are very difficult to optimize, and the efficiency of the join implementation itself becomes crucial to performance. In addition, the application program is entirely responsible for converting the retrieved tuples into the complex graph structure that is required for internal processing. All this results in a significant amount of performance overhead.

In an OODBMS, these complex objects would be directly represented as objects in the OODBMS, with the individual components of these structures also represented as objects. The interrelationships between these objects would be represented by references (in the form of object identifiers or direct in-memory pointers) to the related objects. Thus, the original OODBMS developers generally argued that what must be optimized in an OODBMS to support these applications is *access to an object given its object identifier or other form of pointer to it* (sometimes referred to as *pointer-chasing*).

Following an in-memory pointer can be done extremely quickly, on the order of a few machine instructions. The OODBMS developers argued that following a pointer would be faster than a join (amortized over the number of connections made by the join), no matter how the join is implemented, (judging by the algorithms currently available). In these applications, the mismatch between the set-oriented capabilities of relational DBMSs and the application requirements is sometimes so great that a new class of DBMS is required; the capabilities cannot be provided by taking a relational DBMS and adding features to it, or by implementing those features in applications built on top of a relational DBMS. This has been verified by experience in using ODBMS products in these applications, e.g., by vendors of CAD, CASE, and related products, after a lack of success with relational DBMSs.

The OODBMS vendors that have targeted this class of applications would not necessarily disagree with the idea that a relational system extended with object features might be useful in other applications. However, they believe that such a system does not address the requirements of this category of applications in terms of expressiveness, integration with the host language and, most importantly, performance. The vendors of some of these specialized ODBMSs would also accept the idea that their products might well be slower than relational database systems for traditional business-oriented applications (at least those that have not been significantly extended with requirements for, e.g., user-defined types). However, they note that they do not intend for their products to compete with relational DBMSs for those applications. They appear to believe, however, that their products might replace relational systems in those cases where the relational systems are used, but in a way that indicates they are not well-adapted to the application (e.g., using BLOBs to store complete designs, or using the relational system only for peripheral functions), or where applications currently supported by relational systems have been extended with additional requirements that suggest the use of object features.

As mentioned already, OODBMSs have already been successfully applied to these CAD, CASE, publishing, and similar applications, and it would make sense for potential DBMS users to look into OODBMS products for these applications. However, it is not necessarily the case that only an OODBMS could be used in these applications. An ORDBMS which provided the appropriate type of implementation could also be used. For example, even though it is an ORDBMS, UniSQL supports a form of data-shipping architecture much like an OODBMS, and might also provide the required performance in these applications. Moreover, as noted earlier, the line that separates this class of applications from others is sometimes difficult to draw. For example, it may also be that many CAD, CASE, etc.

applications could also use ODBMSs that take a less-specialized approach. This would depend on the particular performance requirements of the application, as well as other considerations. Also, performance is not the only consideration in many of these applications. Certainly ease of porting existing code to the new environments, and software development life cycle considerations, are also important issues.

While much of the discussion of these ODBMSs has involved their use in CAD and other specialized applications, there are also other applications of these products. For example, an application for this class of ODBMS might be the construction of specialized information services using these ODBMSs as components, e.g. in areas such as financial analysis or telecommunications involving certain specialized kinds of data, or as front-ends for legacy relational systems in client/server architectures, as illustrated by the applications described in Section 7.1.

Also, since many ODBMSs in this class now provide query facilities, the tradeoffs are more complicated than they once were. For example, in applications where generalized querying can be largely replaced by structure traversals and/or less complex queries, such ODBMSs might be candidates for these applications as well.

7.2.2 Applications Requiring Only Object Features

The applications in this category require object features, but are less-dependent than those in the previous category on specific implementation techniques for those features. Depending on the details of the application, these applications could use either one of the specialized ODBMSs described above, an ORDBMS built from the ground up, such as Illustra, or an ORDBMS built on top of a relational system, such as OpenODB/Odapter. One class of applications in this category are applications in which the primary concern is dealing with the impedance mismatch between the highly-structured data required by the application and relational DBMS tables. In some cases, this concern over impedence mismatch is not primarily concern over *performance* (i.e., inefficiencies involved in performing the conversion between application structures and relational tables, or in doing pointer-chasing over the application structures), which might place the application in the previous category, but rather has to do with a simple desire to avoid the development complexity, time, and cost involved in dealing with the impedance mismatch within the application. Another class of applications in this category are applications which could be characterized as conventional relational applications, but which are enhanced with extended data types such as images, text, voice, or video. In these applications, the use of built-in object-oriented features in an ODBMS could be considerably simpler and more effective than trying to perform the same functions using a combination of features (e.g., BLOBs, stored procedures) from even a modern relational DBMS.

Obviously, the specific details of the features required by the application play the major role in the choice of DBMS to support them. Examples of application characteristics that might be primary selection criteria include:

- The need to support complex structures (e.g., in multimedia applications).

- The need for specific extended transaction types (e.g., a checkin/checkout mechanism for long transactions).

- The need to manage a large number of user-defined procedures (to provide specialized behavior for extended data types), or to reuse them via an inheritance mechanism.

- The need for versioning or configuration management facilities.

Such requirements could be supported by either an OODBMS or an ORDBMS providing the specific features required. Primary issues involved in assessing this class of applications include tradeoffs involving the importance of efficient query processing and conformance with current industry standards (such as SQL; although most ODBMSs provide some form of SQL interface, some are much closer to the standard than others), the potential that ODBMSs provide for improved efficiency due to a reduction in mismatches between program and DBMS data types, as well as software engineering and ease-of-use considerations. If highly-optimized query processing is still a substantial part of the application, this might suggest the use of an ORDBMS.

Many ODBMS vendors appear to believe that this type of application, in which there is apt to be more direct competition between ODBMSs and relational DBMSs, will rapidly become the most important market for their products. OODBMS products that attempt to address this area will have to include those facilities that users have come to expect with relational products, and most OODBMSs have made considerable progress in this direction. This market also explains the movement of relational systems in the direction of including more object facilities. The OpenODP/Odapter and Persistence products described in Section 4 illustrate a near-term approach to this direction, in which a "born-again" ODBMS might be created by building an object model processor on top of a relational "engine".

An interesting class of applications in this category are applications that require a front-end integration tool to combine conventional applications requiring relational DBMSs with other systems that require object capabilities. These applications describe the situation faced by, for example, large companies that must manage efforts involving both large amounts of unconventional data (e.g., CAD data, facilities data, geographic data) and large amounts of conventional data, but have not integrated the unconventional data into the databases managing the conventional data due to the difficulties of applying relational DBMS technology to the new data types and structures required for the unconventional data. While the problem is most apparent in industries, such as aerospace, having large amounts of design data, most large companies of *any* kind have large amounts of "unusual" data not currently integrated into DBMSs for similar reasons. Examples include facilities (e.g., outside plant in telecommunications) and geographic information of various kinds (and many companies tend to underestimate the amount of engineering data they actually have). In these environments, a possible integration approach is to continue to actively use relational DBMSs for the more conventional applications (at least until ODBMSs prove themselves in these applications), and build front-ends using ODBMSs as integration tools to provide access to both the new and old data. An example of this type of approach was the Continuum Workstation Platform/Enterprise Solution (CWP/ES) product described in Section 7.1.2. Some general techniques that might be used in doing such integration are described in [CL88, Man88]. Most OODBMS vendors support gateways to relational systems that could be used in implementing this approach. The UniSQL/M product described in Section 4.3 might also be applicable in this type of environment. ODBMSs provide a good basis for an integrated system because their object facilities provide the necessary flexibility in representing many different types of data and data structures, and their associated behavior. This same flexibility also makes ODBMSs a potential basis for new integrated information service applications.

7.2.3 Applications Using an Object-Oriented Development Approach

The applications in this category are those in which software engineering considerations suggest an object-oriented approach to system development, even if the DBMS processing could be viewed as being primarily relational (i.e., processing set-oriented queries over large sets of data with regular structure). In this case, the primary issue may not be the type of data being supported, or the need for specific object types, but simply that the application will be developed in an object-oriented programming language, and the DBMS must provide a smooth match with that language. These applications could use either an OODBMS or an ORDBMS (including such products as the Odapter or Persistence products described in Section 4.4). A primary consideration in choosing a particular ODBMS for applications in this category would obviously be the details of the language interface(s) provided by the product. These details would include not just whether an interface for the particular programming language in question existed, but, e.g., how seamless the interface was.

A number of the applications described in Section 7.1 were at least partially of this type, and this category can be expected to increase as object-oriented software development in general becomes more popular. In addition, once implemented, these applications may well evolve in the direction of one of the other categories, as advantage is taken of the capabilities of objects to represent more complex data types in extending the applications.

7.3 Discussion

The categories of applications described above are only intended to be suggestive of ways in which ODBMSs might be used. Without more experience, it might well be debated what categories of applications actually exist, how to match various types of DBMSs to the various categories, etc. Also, any categorization might change as applications and technology evolve. For example, relational DBMSs took 15 years to mature into efficient systems. If ODBMSs continue to improve in their support for both object-oriented abstractions and relational query processing capabilities, the categories above could become irrelevant.

In the near term, the applicability of ODBMSs in various application environments, where it has not already been established by production use, will be determined by experimentation in small-scale applications. Such experimentation is on-going in many companies and government agencies. The discussion here has attempted to indicate that it is important to pay close attention not only to the types of data to be stored in the DBMS, but also to the details of how the DBMS will be used, the software development environment in which it will be used, and the system life cycle, before determining that an ODBMS is or is not appropriate, and if it is appropriate, what *kind* of ODBMS is appropriate. Ultimately, there is no question about the applicability of ODBMSs, due to the continuing merger of object and relational capabilities in both ODBMS and relational DBMS products and standards. Moreover, as discussed in Section 5, the relational (SQL) and OODBMS (ODMG) standards themselves are moving closer together (and could conceivably merge at some point). As a result, ultimately the issue in selecting a DBMS will be which category of ODBMS is most appropriate for a given application.

A more interesting issue is whether, once the merger on the basis of object *features* has taken place, more than one class of ODBMS based on *implementation* differences will continue to exist, due to the inherent impossibility of merging the required capabilities in a single ODBMS, or whether a merger will eventually take place even here. There are two aspects to this issue: one involves *technology*, the other involves *terminology*.

Some of the *technical* issues have already been mentioned. A number of the OODBMS vendors believe that CAD and conventional data processing applications have such different requirements as to require two classes of systems. That is, although a relational database "engine" can be enhanced with object features, the kind of ODBMS required by the highest-performance CAD-like applications requires a totally different underlying "engine". For example, the ODBMS implementation[22] would be optimized for pointer-chasing and certain forms of caching that would be detrimental to the performance of a basically-relational system. This is not the sort of thing that can be changed once the DBMS has been built. This is similar to the argument that was used by proponents of relational DBMSs against network DBMSs: that the network DBMSs might be made to *look* relational by putting query facilities on top, but they would lack the underlying relational "engine" that would provide the required performance for the queries.

However, while this is an argument why "add-ons" to a basically-relational system cannot be expected to satisfy the requirements of the highest-performance CAD-like applications in the near- to mid-term, it does not necessarily preclude the ultimate creation of ODBMSs that combine underlying implementation support for both efficient pointer-based access to individual objects and efficient query processing over large sets of objects. In fact, an object-oriented approach would appear to provide an ideal development basis for such a system, since the use of objects provides the necessary flexibility to incorporate the various implementation techniques that may be required.

The ability to construct such systems will, however, depend on advances in query optimization, concurrency control, and other technologies. For example, current relational query optimizers have a difficult-enough job optimizing queries over the limited class of storage structures provided by current relational systems. The additional options (such as embedded object pointers and clustering) found in ODBMSs further complicates the task of determining the most efficient query plan. Research currently being conducted on extensible query optimizers (see Section 6.4.3) should directly help in the integration of these classes of systems.

The *terminology* issue basically involves what the resulting systems will be *called*: "object/relational" or "object-oriented" (or even, ultimately, "relational"). A number of the aspects of this issue were discussed in Section 2.11. As noted in that section, at the moment there are no generally-accepted criteria defining what it means, for example, to be "object/relational". Moreover, it is not clear whether such criteria should be based primarily on the externally-visible capabilities provided by the DBMS, primarily on the capabilities of the underlying DBMS "engine" (i.e., on aspects of the DBMS's implementation), or some combination of the two. However, it is safe to say that, as long as there are ODBMSs that differ in some substantial ways, there will be terminology (not necessarily *consistent* terminology) developed to distinguish them.

---

[22] Note that we are not talking about the implementation of individual user-defined objects here, but rather about the implementation of the ODBMS itself, e.g., the choice among the various ODBMS architectures described in Section 6.2, and the implementation of those client and server components.

8. Concluding Remarks

This Technical Report has attempted to describe some of the application requirements that originated the development of ODBMS technology, the basic principles of ODBMSs, and the features of a number of representative ODBMSs (including both OODBMSs and ORDBMSs) that illustrate the directions the technology is taking. The appearance of both OODBMSs and ORDBMSs indicate a general convergence on the *object* concept as a key mechanism for extending DBMS capabilities, even if there is some debate about the way in which other capabilities should be supported. The report has also attempted to describe some of the key issues that may affect the further development of ODBMS technology, including both standards and implementation issues, as well as issues that affect selection of ODBMS technology by users.

ODBMSs are attractive because of their ability to support applications involving complex and widely-varying data types, such as graphics, voice, and text, which are not handled adequately by relational DBMSs. As indicated by the discussion in Section 7, ODBMS applications have in the past primarily focused on specialized applications, such as mechanical and electrical CAD, geographic information systems (GISs), and CASE. However, ODBMSs are receiving increasing attention in other areas, such as telecommunications, financial services, and computer-aided publishing, as well as in conventional data processing applications where an object-oriented development approach is attractive.

In the near term, the applicability of ODBMSs in various application environments, where it has not already been established by production use, will be determined by experimentation in small-scale applications. Such experimentation is on-going in many companies and government agencies. This report has attempted to indicate that it is important to pay close attention not only to the types of data to be stored in the DBMS, but also to the details of how the DBMS will be used, the software development environment in which it will be used, and the system life cycle, before determining that an ODBMS is or is not appropriate. However, ODBMSs definitely represent the next generation of DBMS technology. Current relational DBMS vendors are clearly moving to incorporate object facilities, and object facilities will be incorporated in future relational DBMS (SQL) standards (while, at the same time, ODBMSs are increasingly incorporating support for SQL or SQL-like facilities). As a result, ODBMS technology should be carefully monitored by DBMS users, whether they currently have specific requirements for ODBMS facilities or not. This monitoring can take the following forms:

- Keep abreast of the literature, particularly articles describing applications of ODBMS technology that are similar to yours, or are in the same industry. Some useful sources are cited in the text.

- Establish contacts with users of ODBMS products and find out their experiences with the products (most vendors will provide contacts in user organizations who have agreed to discussions of this type).

- Establish contacts with research or development programs involving ODBMS technology, either in universities or in industry.

- If feasible, obtain copies of selected products and prototype applications that illustrate critical aspects of your requirements.

Acknowledgements

I would like to thank Michael Brodie for his comments on an earlier draft of this report, Gail Mitchell (GTE Laboratories) for her contributions to the material in Section 5, Farshad Nayeri (GTE Laboratories) for discussions of the material in Section 7, and Len Gallagher (NIST), Phil Shaw (Oracle), Bill Kelley (UniSQL), and Amelia Carlson (Sybase) for their help with SQL3 concepts. Any errors or omissions, of course, are my responsibility.

9. References

[AB88] S. Abiteboul and C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects", INRIA Technical Report No. 846, 1988.

[ABDD+89] Malcolm P. Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, Stanley Zdonik, "The Object-Oriented Database System Manifesto", in [KNN89].

[AGOO86] A. Albano, G. Ghelli, M. E. Occhiuto, and R. Orsini, "A Strongly Typed, Interactive Object-Oriented Database Programming Language", in [DD86].

[AH87] Timothy Andrews and Craig Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment", in [Mey87].

[Ahm91] Shamim Ahmed, et. al., "A Comparison of Object-Oriented Database Management Systems for Engineering Applications", Research Report R91-12, Intelligent Engineering Systems Laboratory, Mass. Inst. of Technology, May 1991.

[AK89] S. Abiteboul and P. C. Kanellakis, "Object Identity as a Query Language Primitive", *Proc. 1989 ACM SIGMOD Conf.*, published as *SIGMOD Record*, 18(2), June 1989.

[AP87] Malcolm P. Atkinson and O. Peter Buneman, "Types and Persistence in Database Programming Languages", *ACM Computing Surveys*, 19(2), June 1987.

[Atw93] Thomas Atwood, "ODMG-93: The Object DBMS Standard", *Object Magazine 3*(3), Sept.-Oct. 1993, 37-44.

[Atw94] Thomas Atwood, "ODMG-93: The Object DBMS Standard, Part 2", *Object Magazine 3*(5), January 1994, 32-37.

[Bar94a] Barry & Associates, *DBMS Needs Assessment for Objects*, Release 2.0, Burnsville, MN, 1994.

[Bar94b] Douglas Barry, "The Importance of Understanding ODBMS Features", *Object Magazine 4*(2), May 1994, 76-77.

[BBBD+88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. "The Design and Implementation of O_2, an Object-Oriented Database System", in [Dit88].

[BCD89] Francois Bancilhon, Sophie Cluet, and Claude Delobel, "A Query Language for the O_2 Object-Oriented Database System", in [HMS89].

[BDK92] F. Bancilhon, C. Delobel, P. Kanellakis, *Building an Object-Oriented Database System: The Story of O_2*, Morgan Kaufmann, San Mateo, Calif., 1992.

[Ber94] Paula Bernier, "Object-Based Technology Pinpoints Telecom", *Telephony*, June 6, 1994, 7-16.

[BG81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, June 1981.

[BK86] F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects," *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1986.

[BKKK87] J. Banerjee, W. Kim, H-J. Kim, and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, San Francisco, CA, May 1987.

[BM93] Elisa Bertino and Lorenzo Martino, *Object-Oriented Database Systems: Concepts and Architectures*, Addison-Wesley, Reading, MA, 1993.

[BMOP+89] Robert Bretl, David Maier, Allan Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, Monty Williams, "The GemStone Data Management System", in [KL89].

[BN94] David Beech and Tin Nguyen, "A Look at Object Language Bindings", SQL3 Working Paper X3H2-94-209, June 5, 1994.

[BOHG+92] A. Buchmann, M. Tamer Özsu, M. Hornick, D. Georgakopoulos, and F. Manola, "A Transaction Model for Active Distributed Object Systems", in A. K. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, San Mateo, CA, 1992.

[BOS91] Paul Butterworth, Allen Otis, Jacob Stein, "The GemStone Object Database Management System", *Comm. ACM 34*(10, Oct. 1991, 64-77.

[Bro81] M. Brodie, "On Modelling Behavioral Semantics of Databases", *Proc. Seventh Intl. Conf. on Very Large Data Bases*, September 1981.

[Bro91] Alan W. Brown, *Object-Oriented Databases: Applications in Software Engineering*, McGraw-Hill, London, 1991.

[BTA90] J. A. Blakeley, C. W. Thompson, and A. M. Alashqur, "Strawman Reference Model for Object Query Languages", in [FT91].

[CADF90] The Committee for Advanced DBMS Function, "Third-Generation Data Base System Manifesto", Memorandum UCB/ERL M90/28, University of California, Berkeley, 9 April 1990.

[Cat94a] R. G. G. Cattell (ed.), *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann, 1994.

[Cat94b] R. G. G. Cattell, *Object Data Management*, Revised Edition, Addison-Wesley, Reading, MA, 1994.

[Cat94c] R. G. G. Cattell, "ODMG-93: A Standard for Object-Oriented DBMSs", *Proc. 1994 ACM-SIGMOD Conf. on Management of Data*, Minneapolis, MN, May 1994.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas Tsatalos, Seth White, and Michael J. Zwilling, "Shoring Up Persistent Applications", *Proc. 1994 ACM-SIGMOD Conf. on Management of Data*, Minneapolis, MN, May 1994.

[CDG+90] Michael J. Carey, David J. DeWitt, Goetz Graefe, David M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Skekita, and Scott L. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview", in [ZM89].

[CDKK85] H.-T. Chou, D. DeWitt, R. Katz, and A. Klug, "Design and Implementation of the Wisconsin Storage System", *Software--Practice and Experience 15*(10), October 1985.

[CDLR89] Sophie Cluet, Claude Delobel, Christophe Lecluse, and Philippe Richard, "Reloop, an Algebra Based Query Language for an Object-Oriented Database System", in [KNN89].

[CDMB89] R.C.H. Connor, A. Dearle, R. Morrison, A. L. Brown, "An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance," in [Mey89].

[CDN93] M. Carey, D. DeWitt, and J. Naughton, "The 007 Benchmark", *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, published as *SIGMOD Record* Vol. 22, No. 2, June 1993.

[CDRS89] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Skekita, "Storage Management for Objects in EXODUS", in [KL89]

[CFZ94] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis, "Fine-Grained Sharing in a Page Server OODBMS", *Proc. 1994 ACM SIGMOD Intl. Conf. on Management of Data*, *SIGMOD Record*, Vol. 23, No. 2, June 1994.

[Che76] [CHEN76] P. Chen, "The Entity-Relationship Model--Toward a Unified View of Data," *ACM Trans. Database Systems, 1*, 1 (March 1976).

[Cho93] Li Chou, "Object-Oriented Database Access from Ada", M.S. Thesis, AFIT/GCS/ENG/93M-01, Air Force Institute of Technology, March 1993.

[CL88] Tim Connors and Peter Lyngbaek, "Providing Uniform Access to Heterogeneous Information Bases," in [Dit88].

[Cod79] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning," *ACM Trans. Database Systems, 4*, No. 4 (December 1979).

[CP93] G. Copeland and M.L. Powell, "IBM/JOSS Merged Persistence Service", OMG presentation, December, 1993.

[CS92] R. Cattell and J. Skeen, "Object Operations Benchmark", *ACM Trans. Database Systems 17*(1), March 1992.

[CT85] James Clifford and Abdullah Uz Tansel, "On an Algebra for Historical Relational Databases: Two Views," *Proc. 1985 ACM-SIGMOD Intl. Conference on Management of Data*, ACM, New York, 1985.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee, "An Efficient Implementation of SELF, A Dynamically-Typed Object-Oriented Language Based on Prototypes," in [Mey89].

[Dav89] John Davis, "An Object Oriented Development Case Study," presentation at GTE Laboratories, August, 1989.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy, "Rules Are Objects Too: A Knowledge Model for An Active Object-Oriented Database System", *Proc. 2nd Intl. Workshop on Object Oriented Database Systems*, 129-143, 1988.

[DD86] Klaus R. Dittrich and U. Dayal, eds. *Proc. 1986 Intl. Workshop on Object-Oriented Database Systems*, Washington, IEEE Computer Society Press, 1986.

[DD88] Joshua Dulh and Craig Damon, "A Performance Comparison of Object and Relational Databases Using the Sun Benchmark," in [Mey88].

[DD94] C. J. Date with Hugh Darwen, *A Guide to the SQL Standard*, Addison-Wesley, Reading, MA, 1994.

[Deu91] O. Deux, et. al., "The O_2 System", *Comm. ACM 34*(10), 34-48, October 1991.

[DFMV90] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *Proc. 16th VLDB Conf.*, Brisbane, Australia, Aug. 1990.

[Dit88] Klaus R. Dittrich, ed., *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, Springer-Verlag, 1988.

[DMBC+87] Umeshwar Dayal, Frank Manola, Alejandro Buchmann, Upen Chakravarthy, David Goldhirsch, Sandra Heiler, Jack Orenstein, Arnon Rosenthal, "Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them," *Proceedings BTW-87 (Datenbanksysteme in Buro, Technik und Wissenschaft),* Darmstadt, April 1987, and in [ZM89].

[DMVS89] R. Dixon, T. McKee, M. Vaughan, P. Schweizer, "A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance," in [Mey89].

[DN66] O. J. Dahl and K. Nygaard, "SIMULA--An Algol-based Simulation Language", *Comm. ACM 9*, 671-678.

[DNSV94] David J. DeWitt, Jeffrey F. Naughton, John C. Shafer, and Shivakumar Venkataraman, "ParSets for Parallelizing OODBMS Traversals: Implementation and Performance", submitted to Third Intl. Conf. on Parallel and Distributed Inf. Systems, 1994.

[DPS86] U. Deppisch, H.-B. Paul, and H.-J. Schek, "A Storage System for Complex Objects," in [DD86].

[EH92] G. C. Everest and M. S. Hanna, "Survey of Object-Oriented Database Management Systems", Univ. of Minnesota, January 1992.

[EKMS+94] Andrew Eisenberg, Krishna Kultarni, Nelson Mattos, Phil Shaw, and David Beech, "_RECORD_TYPEs - base proposal", SQL3 Change Proposal X3H2-94-103 and ISO/IEC JTC1/SC21 WG3 DBL proposal SOU-076, March 23, 1994.

[Eng92] Larry English, "Object Databases At Work", *DBMS*, Vol. 5, No. 11, October 1992, 44-58.

[ES94] Eric Eldred and Tim Sylvester, "Two Databases for Modeling Complex Data Without the Oh-Ohs", *Client/Server Today*, 67-72, May 1994.

[FACC+89] D. H. Fishman, J. Annevelink. E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, W.K Wilkinson, "Overview of the Iris DBMS", in [KL89].

[FBCC+87] D. H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.D. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan, "Iris: An Object-Oriented Database Management System", *ACM Trans. Office Inf. Systems, 5*(1), January 1987.

[Feu89] Mike Feuche, "Object-Oriented Databases Finding Applications," Management Information Systems Week, April 10, 1989.

[Fin93] Richard Finkelstein, "Breaking the Mold", *Database Programming & Design*, Vol. 6, No. 2, February 1993.

[FKMT91] Elizabeth Fong, William Kent, Ken Moore, and Craig Thompson (eds), "X3/SPARC/DBSSG/OODBTG Final Report," 17 September, 1991, National Institute of Standards and Technology, 1991.

[FT83] P. Fischer and S. Thomas, "Operators for Non-First-Normal-Form Relations," *Proc. COMPSAC83*, November, 1983.

[FT91] Elizabeth N. Fong and Craig W. Thompson (eds.), "Proceedings of the Object-Oriented Database Task Group Workshop, Tuesday, May 22, 1990, Atlantic City, NJ", NISTIR 4503, National Institute of Standards and Technology, February 1991.

[Gal94] Leonard Gallagher, personal communications, July 1994.

[Gar94] Greg Garry, "How the Object-Oriented World Translates Into Reality", Client/Server Today, May 1994, 56-63.

[GCDM+93] Goetz Graefe, Richard L. Cole, Diane L. Davison, William J. McKenna, and Richard H. Wolniewicz, "Extensible Query Optimization and Parallel Execution in Volcano", in J. C. Freytag, D. Maier, and G. Vossen (eds), *Query Processing for Advanced Database Systems*, Morgan Kaufmann, San Mateo, CA, 1993.

[GD87] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator", *Proc. 1987 SIGMOD Intl. Conf. on Management of Data*, San Francisco, CA, May 1987.

[GH92] D. Georgakopoulos and M. Hornick, "An Environment for Specification and Management of Extended Transactions and Workflows in DOMS", TR-0218-09-92-165, GTE Laboratories Incorporated, October 30, 1992.

[GM87] G. Graef and D. Maier, "Query Optimization in Object-Oriented Database Management Systems: A Prospectus," *Proceedings BTW-87 (Datenbanksysteme in Buro, Technik und Wissenschaft),* Darmstadt, April 1987.

[GR83] Adele Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[GS91] Leonard Gallagher and Joan Sullivan, "Database Language SQL: Integrator of CALS Data Repositories", National Institute of Standards and Technology, November 27, 1991

[Hal93] Timothy J. Halloran, "Performance Measurement of Three Commercial Object-Oriented Database Management Systems", M.S. Thesis, AFIT/GCS/ENG/93D-12, Air Force Institute of Technology, December 1993.

[Haw94] Paula Hawthorn, personal communication, June 15, 1994.

[HFLP89] Laura M. Haas, J. C. Freytag, Guy Lohman, and Hamid Pirahesh, "Extensible Query Processing in Starburst", *Proc. 1989 ACM SIGMOD Intl. Conf. on Management of Data*, 377-388, June 1989.

[HG93] M. Hornick and D. Georgakopoulos, "Towards Addressing the Limitations of Conventional Transactional Systems in Supporting Advanced Applications", TR-0241-12-93-165, GTE Laboratories Incorporated, December 20, 1993.

[HL82] R. Haskin and R. Lorie, "On Extending the Functions of a Relational Database System," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1982.

[HLM88] M. Hsu, R. Ladin, and D. McCarthy, "An Execution Model for Active Data Base Management Systems", *Proc. 3rd. Intl. Conf. on Data and Knowledge Bases*, June 1988.

[HM78] M. Hammer and D. McLeod, "The Semantic Data Model," *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, 1978.

[HMS89] R. Hull, R. Morrison, and D. Stemple (eds.), *Database Programming Languages, Proc. 2nd Intl. Workshop*, Morgan Kaufmann, San Mateo, 1989.

[HP93] Hewlett-Packard Company, *OpenODB Reference Document*, Part No. B3768-90001, 1993.

[HZ87] Mark F. Hornick and Stanley B. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database, *ACM Trans. Office Information Systems*, 5(1), 70--95, January 1987.

[Ing86] D. H. H. Ingalls, "A Simple Technique for Handling Multiple Polymorphism", in [Mey86].

[Int93] *MATISSE Product Overview*, Intellitic International, Inc., 1993.

[Ita93] Itasca Systems, Inc., *ITASCA Distributed Object Database Management System, Technical Summary for Release 2.2*, 1993.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski, "TS: An Optimizing Compiler for Smalltalk," in [Mey88].

[KA90] Setrag Khoshafian and Razmik Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces*, John Wiley, New York, 1990.

[Kae86] T. Kaehler, "Virtual Memory on a Narrow Machine for an Object-Oriented Language", in [Mey86].

[Kay94] Russell Kay, "Objects in Use", *Byte*, April 1994, 99-104.

[KBC+88] Won Kim, N. Ballou, Hong-Tai Chou, J.F. Garza, D. Woelk, and J. Banerjee, "Integrating an Object-Oriented Programming System with a Database System", in [Mey88].

[KBCG89] Won Kim, Nat Ballou, Hong-Tai Chou, and Darrell Garza, Jorge F. Woelk, "Features of the ORION Object-Oriented Database System", in [KL89].

[KBGW91] Won Kim, N. Ballou, J.F. Garza, and D.; Woelk, "A Distributed Object-Oriented Database System Supporting Shared and Private Databases", *ACM Trans. Inf. Syst.*, 9(1):31--51, January 1991.

[KCS94] William Kelley, Amelia Carlson, Phil Shaw, personal communication, August 1994.

[Kel94a] William Kelley, "Row Identifiers Simplify Applications", SQL3 Change Proposal X3H2-94-4, January 31, 1994.

[Kel94b] William Kelley, "Path Expressions for Row Identifiers", SQL3 Change Proposal X3H2-94-90, March 16, 1994.

[Kel94c] William Kelley, "Defining Extents for ADT's", SQL3 Change Proposal X3H2-94-312, July 11, 1994.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Woelk, "Architecture of the ORION Next-Generation Database System", *IEEE Trans. Knowledge and Data Engrg.*, 2(1):109--24, March 1990.

[Kho87] S. Khoshafian, et. al., "A Query Processing Algorithm for the Decomposed Storage Model," *Proc. Intl. Conf. on Data Engineering,* 1987.

[Kho93] Setrag Khoshafian, *Object-Oriented Databases*, John Wiley, New York, 1993.

[Kil92] Michael F. Kilian, *Parallel Sets: An Object-Oriented Methodology for Massively Parallel Programming*, Ph.D. Thesis, Havard Center for Research in Computing Technology, Cambridge, MA, 1992.

[Kim94] Won Kim, "UniSQL/X Unified Relational and Object-Oriented Database System", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, *SIGMOD Record*, Vol. 23, No. 2, June 1994.

[Kim94a] Won Kim, "Observations on the ODMG-93 Proposal for an Object-Oriented Database Language", *SIGMOD Record* 23(1), March 1994, 4-9.

[Kim94b] Won Kim, "Observations on ODMG-93", *OMG First Class*, Vol. IV, Issue II, March/April 1994, 14-15.

[KJA94] Arthur Keller, Richard Jensen, and Shailesh Agarwal, "Bridging the Gap Between C++ and Relational Databases", in *Persistence Technical Overview*, Persistence Software, Inc., January 4, 1994.

[KK94] William Kelley and Won Kim, "Observations on the Current SQL3 Object Model Proposal (and Inviation for Scholarly Opinions)", X3H2-94-58, UniSQL, Inc., January, 1994.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv, "Querying Object-Oriented Databases", *Proc. ACM SIGMOD Conf.*, 1992.

[KL89] Won Kim and Frederick H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications*, ACM Press / Addison-Wesley, 1989.

[KNN89] Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, eds., *Proc. First Intl. Conf. on Deductive and Object-Oriented Databases*, Elsevier Science Publishers, Amsterdam, 1989.

[KRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, 1991.

[KT91] Yutaka Kimura and Kunitoshi Tsuruoka, "A Language and View for Odin, an Object-Oriented Database System", *Proc. IEEE Pacific Rim Conf. on Comm., Comput., and Signal Proc.*, 284-287, May 1991.

[Kul93] Krishna G. Kulkarni, "Object-Orientation and the SQL Standard", *Computer Standards & Interfaces 15* (1993), 287-300.

[Lis81] B. Liskov, *CLU Reference Manual,* Lecture Notes in Computer Science 114, Springer-Verlag, 1981.

[LK86] Peter Lyngbaek and William Kent, "A Data Modeling Methodology for the Design and Implementation of Information Systems", in [DD86], 6-17.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb, "The ObjectStore Database System", *Comm. ACM 34*(10), 50-63 (October 1991).

[LMSV+93] T. W. Leung, G. Mitchell, B. Subramanian, B. Vance, S. L. Vandenberg, and S. B. Zdonik, "The AQUA Data Model and Algebra", *Proc. Fourth Intl. Workshop on Database Programming Languages* (1993), 157-175.

[Loc85] F. Lochovsky (ed.), *Database Engineering*, Vol. 8, No. 4, Special Issue on Object-Oriented Systems, IEEE Computer Society, 1985.

[Loo92] Mary E. S. Loomis, "Client-Server Architecture", *Journal of Object-Oriented Programming 4*(9), February 1992, 40-44.

[Loo94] Mary E. S. Loomis, "ODBMS Myths and Realities", *Journal of Object-Oriented Programming 7*(4), July/August 1994, 77-80.

[LP83] R. Lorie and W. Plouffe, "Complex Objects and their Use in Design Transactions," *Proc. ACM SIGMOD Database Week -- Engineering Design Applications*, May 1983.

[LRV88] C. Lecluse, P. Richard, and F. Velez. "O_2, an Object-Oriented Data Model", *Proc. ACM/SIGMOD Intl. Conf. on Management of Data*, 1988 (also in [ZM89]).

[LV87] Peter Lyngbaek and Victor Vianu, "Mapping a Semantic Database Model to the Relational Model", *Proc. ACM-SIGMOD 1987 Annual Conf.*, ACM, New York, 1987.

[LW89] Peter Lyngbaek and Kevin Wilkinson, "An Overview of the Iris Kernel Architecture", *Object-Oriented Programming Systems*, Pitman Publishing, London, 1989.

[Mai89] David Maier, "Why Isn't There an Object-Oriented Data Model", in Ritter (ed.), *Information Processing 89*, IFIPS North-Holland, 1989.

[Man88] Frank Manola, "Distributed Object Management Technology", TM-0014-06-88-165, GTE Laboratories Incorporated, June 30, 1988.

[Man89a] Frank Manola, "An Evaluation of Object-Oriented DBMS Developments," TR-0066-10-89-165, GTE Laboratories Incorporated, October 31, 1989.

[Man89b] Frank Manola, "Object Model Capabilities for Distributed Object Management," TM-0149-06-89-165, GTE Laboratories Incorporated, June 30, 1989.

[Man93] Frank Manola, "MetaObject Protocol Concepts for a 'RISC' Object Model," Technical Report TR-0244-12-93-165, GTE Laboratories Incorporated, December, 1993.

[Man94] Frank Manola (ed.), X3H7 Object Model Features Matrix, Document No. X3H7-93-007v8, May 2, 1994.

[MBW80] J. Mylopoulos, P. Bernstein, and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. Database Systems, 5*, 2, (June 1980).

[MD89] Frank Manola and Umeshwar Dayal, "PDM: An Object-Oriented Data Model", in [DD86] and in [ZM89].

[MD93] Nelson Mattos and Linda DeMichiel, "Why We Should Remove Table of ADTs from SQL3", ISO/IEC JTC1/SC21/WG3 DBL YOK-080, ANSI X3H2-169rev1, March 29, 1993.

[MDZ93] Gail Mitchell, Umeshwar Dayal, and Stanley B. Zdonik, "Control of an Extensible Query Optimizer: A Planning-Based Approach", *Proc. 19th VLDB Conf.*, 517-528, August, 1993.

[Mel94] Jim Melton (ed.), ISO-ANSI Working Draft, Database Language SQL/Foundation (SQL3), March 1994.

[Mer84] T. H. Merrett, *Relational Information Systems*, Reston Publishing Co., Reston, 1984.

[Mey86] N. Meyrowitz, ed., *OOPSLA '86 Conference Proceedings*, ACM, Sept., 1986, published as *SIGPLAN Notices*, 21(11), Nov., 1986.

[Mey87] N. Meyrowitz, ed., *OOPSLA '87 Conference Proceedings*, ACM, Oct., 1987, published as *SIGPLAN Notices*, 22(12), Dec., 1987.

[Mey88] N. Meyrowitz, ed., *OOPSLA '88 Conference Proceedings*, ACM, Sept., 1988, published as *SIGPLAN Notices*, 23(11), Nov., 1988.

[Mey89] N. Meyrowitz, ed., *OOPSLA '89 Conference Proceedings*, ACM, Oct., 1989, published as *SIGPLAN Notices*, 24(10), Oct., 1989.

[MHB90] Frank Manola, Mark F. Hornick, and Alejandro P. Buchmann, "Object Data Model Facilities for Multimedia Data Types", Technical Memorandum TM-01332-11-90-165, GTE Laboratories Incorporated, December, 1990.

[MM94] Frank Manola and Gail Mitchell, "A Comparison of Candidate Object Models for Object Query Services", X3H7-94-32v1, August 28, 1994.

[Mon93] Montage Software, Inc., *Montage User's Guide*, Montage DBMS Release 1.2, December 1993.

[Moy93] Anthony Moyers, "An Object-Oriented Database Interface for Ada", M.S. Thesis, AFIT/GCS/ENG/93D-11, Air Force Institute of Technology, December 1993.

[MR93] Karl S. Mathias and Mark A. Roth, "An Examination of Two Ada Language Object-Oriented Databases", AFIT/EN-TR-93-6, Air Force Institute of Technology, 8 July 1993.

[MS93] Jim Melton and Alan R. Simon, *Understanding the New SQL: A Complete Guide*, Morgan Kaufmann, San Francisco, 1993.

[MSOP86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy, "Development of an Object-Oriented DBMS", in [Mey86].

[MZ89] David Maier and Stanley Zdonik, "Fundamentals of Object-Oriented Databases", in [ZM89].

[MZD92] Gail Mitchell, Stanley B. Zdonik, and Umeshwar Dayal, "An Architecture for Query Processing in Persistent Object Stores", *Proc. Hawaii Intl. Conf. on System Sciences*, 787-798, January 1992.

[Obj93] Objectivity, Inc., *Objectivity Technical Overview*, 1993

[OBS86] P. O'Brien, B. Bullis, and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl", in [DD86].

[ODI92a] Object Design, Inc., *ObjectStore Tutorial*, Release 2.0, October 1992.

[ODI92b] Object Design, Inc., *ObjectStore User Guide*, Release 2.0, October 1992.

[ODI93] Usenet newsgroup comp.databases.object, miscellaneous messages from ODI employees, August 1993.

[ODI94] Object Design, Inc., *ObjectStore Technical Overview*, Release 3, March 1994.

[OFAQ93] Usenet newsgroup comp.object "Frequently Asked Questions (FAQ) List and Available Systems For Object-Oriented Technology", Version 1.0.2 (9-93) Part 2/7, R. Hathaway, Sept. 9, 1993.

[Ols94] Tom Olson, *The OOT Info Exchange*, Vol. 4, No. 3, GTE Corporate Information Technology Asset Management, Tampa, FL, July 1994.

[OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Rev. 1.1, 1991.

[OMG92a] Object Management Group, *Object Management Architecture Guide*, Revision 2.0, 2nd. Ed., OMG TC Document 92.11.1, Sept. 1, 1992.

[OMG92b] Object Management Group, *Object Services Architecture*, Revision 6.0, OMG Document 92.8.4, 28 August 1992.

[OMG94a] Object Management Group, *Common Object Services Specification, Volume I*, Revision 1.0, OMG Document Number 94-1-1, March 1, 1994.

[OMG94b] Object Management Group, *Object Services RFP4*, OMG TC Document 94.3.8, March 28, 1994.

[ONT94a] ONTOS, Inc., "Introduction to ONTOS DB 3.0", ONT-30-SUN-IODB-1.0, April 1994.

[ONT94b] ONTOS, Inc., "ONTOS DB 3.0 Object SQL Guide", ONT-30-SUN-OSQLG-1.0, April 1994.

[Ore89] Jack Orenstein, UseNet news message, October 6, 1989.

[Per94] Persistence Software Quarterly Newsletter No. 2, July 1994.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst", *Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data*, 39-48, June 1992.

[Pir94] Hamid Pirahesh, "Object-Oriented Features of DB2 Client/Server", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, *SIGMOD Record*, Vol. 23, No. 2, June 1994.

[RC89] Joel E. Richardson and Michael J. Carey, "Persistence in the E language: Issues and implementation", *Software -- Practice and Experience*, 19(12):1115--1150, December 1989.

[Ric92] Mike Ricciuti, "The Easy Way to OOPS", *Datamation*, February 1, 1992.

[RS79] L. A. Rowe and K. A. Shoens, "Data Abstraction, Views, and Updates in RIGEL", *Proc. 1979 ACM-SIGMOD Intl. Conference on Management of Data*, ACM, New York, 1979.

[SB86] M. J. Stefik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations", *The AI Magazine*, 6(4), 1986.

[Sch77] J. W. Schmidt, "Some High Level Language Constructs for Data of Type Relation", *ACM Trans. Database Systems*, 2(3), Sept. 1977.

[SFL83] J. M. Smith, S. A. Fox, and T. A. Landers, "ADAPLEX: Rationale and Reference Manual", Technical Report CCA-83-08, Computer Corporation of America, May 1983.

[Sha94] Phil Shaw, "OID Association", X3H2 proposal X3H2-94-268, 3 June 1994.

[Shi81] D. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. Database Systems, 6*, 1, (March 1981).

[Sny86] Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", in [Mey86].

[SR86] Michael Stonebraker and Larry Rowe, "The Design of POSTGRES", *Proc. 1986 ACM-SIGMOD Intl. Conference on Management of Data*, ACM, New York, 1986.

[SS86] H.-J. Schek and M. H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems*, 11,4, 1986.

[SS90] Edward Sciore and John Sieg, Jr., "A Modular Query Optimizer Generator", *Proc. 6th Intl. Conf. on Data Engrg.*, 146-153, 1990.

[Ste90] G. L. Steele, Jr., *Common Lisp: The Language* (2nd. Edition), Digital Press, Bedford, MA, 1990.

[Sto86a] Michael Stonebraker, "Object Management in POSTGRES Using Procedures", in [DD86].

[Sto86b] Michael Stonebraker, "Inclusion of New Types in Relational Data Base Systems", *Proc. Intl. Conf. on Data Engineering*, IEEE, 1986.

[Str86] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1986

[Sut93] J. Sutherland, "Matisse OODBMS Object Model", ANSI X3H7-93-058, 28 May 1993.

[SW87] Bruce Shriver and Peter Wegner, eds., *Research Directions in Object-Oriented Programming*, MIT Press, 1987 (early versions of some of these papers appeared in *SIGPLAN Notices*, 21(10), October 1986).

[SZ87] Andrea H. Skarra and Stanley B. Zdonik, "Type Evolution in an Object-Oriented Database", in [SW87].

[SZ89] Andrea H. Skarra and Stanley B. Zdonik, "Concurrency Control for Cooperating Transactions in an Object-Oriented Database, in [KL89].

[SZ89b] Gail M. Shaw and Stanley B. Zdonik, "An Object-Oriented Query Algebra", in [HMS89].

[TI93] Texas Instruments, Inc., *Open OODB Technical Overview Release 0.2 (Alpha)*, 1993.

[TKN92] Kunitoshi Tsuruoka, Yutaka Kimura, and Misa Namiuchi, "Architecture of an Object-Oriented Database Management System - Odin", *NEC Research & Development 33*(4), 669-678 (October 1992).

[Tsu93] Kunitoshi Tsuruoka, "Features of an Object-Oriented DBMS - PERCIO", seminar given at GTE Laboratories, November, 1993.

[Ube94] Michael Ubell, "The Montage Extensible DataBlade Architecture", *Proc. ACM-SIGMOD Intl. Conf. on Management of Data*, *SIGMOD Record*, Vol. 23, No. 2, June 1994.

[Uni91] UniSQL, Inc., *UniSQL/X Database Management System Product Description*, 1991.

[Var93] Sanjeev Varma, "Objects and Databases: Where Are We Now?", *Database Programming & Design*, Vol. 6, No. 5, May 1993.

[Vel93] Fernando Velez, "Object-Oriented Database Systems: Implementation Issues", Tutorial, 1993 ACM SIGMOD Intl. Conf. on Management of Data.

[Ver93a] Versant Object Technology, *VERSANT Object Database Management System C++/VERSANT Manual*, Release 2, July 1993.

[Ver93b] Versant Object Technology, *Objectworks\Smalltalk/VERSANT Reference*, VERSANT Release 1.10, March 1993.

[Ver93c] Versant Object Technology, *VERSANT Object SQL, VERSANT C++ Application ToolSet Version 2.0*, June 1993.

[Ver93d] Versant Object Technology, *VERSANT Object Database Management System System Manual*, Release 2, July 1993.

[WBT92] David L. Wells, Jose A. Blakeley, and Craig W. Thompson, "The Open Object-Oriented Database: Obtaining Database Functionality by Extension", *IEEE Computer* 25(10), Oct., 1992.

[Weg87] P. Wegner, "Dimensions of Object-Based Language Design", in [Mey87].

[Weg90] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming", *OOPS Messenger*, 1(1), ACM, August 1990.

[Woy93] Mark Woyna, et.al., "Modeling Battlefield Sensor Environments with an Object Database Management System", *ACM SIGMOD Record, 22*(3): 499-501 (June 1993).

[Woy94] Mark Woyna, "Modeling Battlefield Sensor Environments with an ODBMS", *Object Magazine 3*(5): 44-46 (January 1994).

[YT87] A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[ZM89] Stanley B. Zdonik and David Maier (eds.), *Readings in Object-Oriented Database Systems*, Morgan-Kaufmann, 1989.