TR-0263-08-94-165

An Evaluation of Object-Oriented DBMS Developments
1994 Edition

Frank Manola

August 31, 1994

GTE LABORATORIES INCORPORATED
40 Sylvan Road
Waltham, MA 02254

TR-0263-08-94-165

This document is a GTE Laboratories Technical Report. It describes results and conclusions reached upon completion of a major phase of a research project. The ideas and views put forth by the author have been reviewed and accepted by the appropriate Technology Center Director(s) or Laboratory Director(s).

______________________________________ ______________

Author **Date**

______________________________________ ______________

Manager **Date**
Distributed Object Computing Department

______________________________________ ______________

Director **Date**
Computer and Intelligent Systems Laboratory

Abstract

Organizations of all sizes are attempting to employ database technology to address increasingly complex application requirements. In some cases, these requirements involve specialized application domains, such as Computer-Aided Design or Manufacturing (CAD/CAM), Computer-Aided Software Engineering (CASE), office automation, multimedia content development and presentation, etc. In other cases, however, these requirements involve more conventional business applications, such as billing, customer/employee record keeping, order processing, or maintenance of outside plant, but with extended requirements of various sorts.

In attempting to apply Database Management Systems (DBMSs) to these applications, it became widely recognized that extensions to conventional relational DBMS technology were required to address the requirements of these applications. The first result of attempts to address these requirements was the development of the *Object-Oriented DBMS* (OODBMS). The OODBMS combines capabilities of conventional DBMSs and object-oriented programming languages such as Smalltalk. A number of reasonably-mature OODBMS products exist, and are widely used in certain specialized applications. Increasingly, these products are being enhanced with query and other facilities similar to those provided by relational DBMSs, and are being applied in more general-purpose applications.

A new class of DBMS, the *Object/Relational DBMS* (ORDBMS), constitutes the most recent piece of the puzzle. ORDBMSs are to some extent the results of research on extending relational DBMSs, but are also based to some extent on the emergence of OODBMSs as viable products. The ORDBMSs attempt to combine the capabilities of an OODBMS and a relational DBMS (and current individual products represent different mixtures of these capabilities). The appearance of ORDBMSs, together with the general support now provided for query facilities (and object extensions to the relational standard SQL language) by OODBMSs, indicates that a merger of OODBMS and relational capabilities is taking place, and that even more complete mergers can be expected as OODBMS and ORDBMS technology matures further. Moreover, the appearance of these two classes of products indicates a general convergence on the *object* concept as a key mechanism for extending DBMS capabilities. As a result, this report refers to both classes of products generally as *Object DBMSs* (ODBMSs).

This report begins by describing the limitations of conventional relational DBMSs that led to the emergence of ODBMS technology. The report then describes the basic concepts of object-oriented software, and the general characteristics to be expected in an ODBMS. It also provides examples of how these ODBMS characteristics apply to aspects of advanced applications. The report then describes a number of representative OODBMS and ORDBMS products that illustrate the state of commercial ODBMS technology. The report also discusses key standardization activities that affect the development of ODBMS technology, including the object extensions to SQL (SQL3) being developed by ANSI and ISO standards committees, proposed standards for OODBMSs defined by the Object Database Management Group, and proposed standards for distributed object systems being developed by the Object Management Group. The report then provides an overview of the different ways that ODBMS products provide key features, and points out some of the key technical issues represented by the various choices. The report also categorizes various classes of ODBMS applications, discusses specific examples of applications, such as telecommunications applications, and provides some overall conclusions that can be reached about ODBMSs. The report also contains an extensive reference list.

Trademark Acknowledgements

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this report, and we are aware of a trademark claim, the designations have been printed in caps or initial caps.

Executive Summary

Organizations of all sizes are increasingly attempting to employ database technology to address increasingly complex application requirements. In some cases, these requirements involve specialized application domains, such as Computer-Aided Design or Manufacturing (CAD/CAM), Computer-Aided Software Engineering (CASE), office automation, multimedia content development and presentation, etc. In other cases, however, these requirements involve more conventional business applications, such as billing, customer/employee record keeping, order processing, or maintenance of outside plant, but with extended requirements of various sorts, for example:

- outside plant information extended with geographic or graphic information about sites where equipment or facilities are located

- insurance claim information extended with photographs of property damage involved in the claim

- general business applications that must be integrated into distributed client/server architectures, or that have graphical user interfaces developed using object-oriented techniques

In attempting to apply Database Management Systems (DBMSs) to these applications, it became widely recognized that extensions to conventional relational DBMS technology were required to address the requirements of these applications. The first result of attempts to address these requirements was the development of the *Object-Oriented DBMS* (OODBMS). The OODBMS combines capabilities of conventional DBMSs and object-oriented programming languages such as Smalltalk. A number of reasonably-mature OODBMS products exist, and are widely used in certain specialized applications. Increasingly, these products are being enhanced with query and other facilities similar to those provided on relational DBMSs, and are being applied in more general-purpose applications.

A new class of DBMS, the *Object/Relational DBMS* (ORDBMS), constitutes the most recent piece of the puzzle. ORDBMSs are to some extent the results of research on extending relational DBMSs, but are also based to some extent on the emergence of OODBMSs as viable products. The ORDBMSs attempt to combine the capabilities of an OODBMS and a relational DBMS (and current individual products represent different mixtures of these capabilities). While many of the OODBMSs originally emphasized their non-relational capabilities, with query facilities "added on" or otherwise considered less than crucial, ORDBMSs are based on the idea of full relational query support as basic, with the addition of objects (or abstract data types) as additional data structure concepts that can be used to generalize relational tables and types used within them. The appearance of ORDBMSs, together with the general support now provided for query facilities (and object extensions to the relational standard SQL language) by OODBMSs, indicates that a merger of OODBMS and relational capabilities is taking place, and that even more complete mergers can be expected as OODBMS and ORDBMS technology matures further. Moreover, the appearance of these two classes of products indicates 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. As a result, this report refers to both classes of products generally as *Object DBMSs* (ODBMSs).

This report describes the motivation for the extended DBMS technology represented by ODBMSs, describes representative ODBMS products and related standardization activities, and discusses the general concepts and facilities of ODBMSs and some of the variants of these that appear in the various products. This report is an update of an earlier report, which surveyed the state of the art of OODBMS technology, and the development of extensions to relational DBMSs, as of 1989. The current report provides updated descriptions of current OODBMS products, and also covers the more recent developments in ORDBMS products, and ODBMS-related standardization activities.

This report does not attempt to be exhaustive in covering all ODBMS products, or in describing the details of any individual products. Moreover, the report does not attempt to "rate" the various systems covered against some standard set of features, or according to some other rating system, in order to choose the "best product". Applications and application contexts differ so widely that it is impossible to do this and be fair to the various products. Potential users of the products should evaluate more detailed information about each product against the requirements of particular applications in order to make product selections.

The report is organized as follows. Section 1 describes the limitations of conventional relational DBMSs that led to the emergence of ODBMS technology. Section 2 describes the basic concepts of object-oriented software, and the general characteristics to be expected in an ODBMS. It also provides examples of how these ODBMS characteristics apply to aspects of advanced applications, and attempts to clarify the distinction made above between OODBMSs and ORDBMSs. (This and subsequent sections also discuss why this distinction may be rather temporary.) Section 3 provides descriptions of a number of representative OODBMS products that illustrate the state of commercial OODBMS technology. Section 4 then provides descriptions of several Object/Relational DBMS (ORDBMS) products. These products represent attempts to extend relational DBMS technology with object capabilities. Section 5 discusses key standardization activities that affect the development of ODBMS technology, including the object extensions to SQL (SQL3) being developed by ANSI and ISO standards committees, proposed standards for OODBMSs defined by the Object Database Management Group, and proposed standards for distributed object systems being developed by the Object Management Group. Section 6 then provides an overview of the different ways that ODBMS products provide key features, and points out some of the key technical issues represented by the various choices. Section 7 describes a categorization of various classes of ODBMS applications, and discusses specific examples of applications, such as telecommunications applications. Finally, Section 8 describes some overall conclusions that can be reached about ODBMSs. The report concludes with an extensive reference list.

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. 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 attempts 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.

Table of Contents

1. Introduction and Background

Organizations of all sizes are increasingly attempting to employ database technology to address increasingly complex application requirements. In some cases, these requirements involve specialized application domains, such as:

- Computer-Aided Design (CAD)
- Computer-Aided Manufacturing (CAM)
- Geographic Information Systems (GIS)
- Computer-Aided Software Engineering (CASE)
- office automation
- multimedia content development and presentation

In other cases, these requirements involve more conventional business applications, such as billing, customer/employee record keeping, order processing, or maintenance of outside plant, but with extended requirements of various sorts, for example:

- outside plant information extended with geographic or graphic information about sites where equipment or facilities are located

- insurance claim information extended with photographs of property damage involved in the claim

- general business applications that must be integrated into distributed client/server architectures, or that have graphical user interfaces developed using object-oriented techniques

In attempting to apply Database Management Systems (DBMSs) to these applications, it became widely recognized that extensions to conventional DBMS technology were required to address the requirements of these applications. Developing the extended DBMS technology to address these requirements has been an extremely active area of both database research and product development activity. The result of attempts to address these requirements has been the emergence of the Object-Oriented DBMS (OODBMS) and, more recently, its extended relational counterpart, the Object/Relational DBMS (ORDBMS). These classes of products, which this report refers to generally as Object DBMSs (ODBMSs), attempt to provide a combination of the flexibility and extensibility associated with object-oriented programming, together with the facilities expected of a generalized DBMS.

The purpose of this report is to describe the motivation for the extended DBMS technology represented by ODBMSs, to describe representative ODBMS products and related standardization activities, and to discuss the general concepts and facilities of ODBMSs and some of the variants of these that appear in the various products. This report is an update of [Man89a], which surveyed the state of the art of OODBMS technology, and the development of extensions to relational DBMSs, as of 1989. The current report provides updated descriptions of current OODBMS products, and also covers the more recent developments in ORDBMS products, and ODBMS-related standardization activities.

This report does not attempt to be exhaustive in covering all ODBMS products, or in describing the details of any individual products. Moreover, the report does not attempt to "rate" the various systems covered against some standard set of features, or according to some other rating system, in order to choose the "best product". Applications and application contexts differ so widely that it is impossible to do this and be fair to the various products. Potential users of the products should evaluate more detailed information about each product against the requirements of particular applications in order to make product selections. More is said on this subject in later sections.

The rest of this report is organized as follows. The rest of Section 1 describes the limitations of conventional relational DBMSs that led to the emergence of ODBMS technology. Section 2 describes the basic concepts of object-oriented software, and the general characteristics to be expected in an ODBMS. It also provides examples of how these ODBMS characteristics apply to aspects of advanced applications, and attempts to clarify the distinction made above between OODBMSs and ORDBMSs. (This and subsequent sections also discuss why this distinction may be rather temporary.) Section 3 provides descriptions of a number of representative OODBMS products that illustrate the state of commercial OODBMS technology. Section 4 then provides descriptions of several examples of a new class of database products, the Object/Relational DBMS (ORDBMS). These products represent attempts to extend relational DBMS technology with object capabilities. Section 5 discusses key standardization activities that affect the development of ODBMS technology, including the object extensions to SQL (SQL3) being developed by ANSI and ISO standards committees. Section 6 then provides an overview of the different ways that ODBMS products provide key features, and points out some of the key technical issues represented by the various choices. Section 7 describes a categorization of various classes of ODBMS applications, and discusses specific examples of applications. Finally, Section 8 describes some overall conclusions that can be reached about ODBMSs. The report also contains an extensive reference list.

1.1 Limitations of Conventional Relational DBMSs

Conventionally, a DBMS is associated with a particular data model, and provides an implementation of that model. In particular, a relational DBMS implements the relational data model. The relational model defines

- a single data structuring concept, the *relation* (or *table*), for representing all data in the database
- a set of operations on relations called the *relational algebra* (these operations include, select, project, join, union, etc.)
- a set of constraints, e.g., values of columns in tables must be atomic

Relational DBMSs then provide:

- a set of *data types* for use in columns (e.g., CHAR, DATETIME, FLOAT)
- a language that can be defined in terms of the algebra (SQL)
- an implementation of the language

Figure 1.1 shows the general pattern of application access in a relational DBMS. The application program sends SQL queries to the DBMS. These queries specify qualifications and other manipulations of the various database tables, and result in other tables (in SQL, tables are technically multisets of rows) being returned to the program. The retrieved rows must be mapped to variables or other data structures in the application's programming language. The program operates on these data structures, possibly producing changes to them. If these changes must be reflected in the database, the program formulates SQL update requests to perform the necessary changes, using the application variables or data structures to prepare the necessary arguments for these update requests. Most of the data semantics reside in the application program, since the ability of the DBMS to represent complex data structures is limited, and the DBMS provides only limited capability to associate particular behavior with the data. (This is so even though many relational DBMSs now support various facilities for storing procedures in the database, and for defining certain types of constraints on database data).

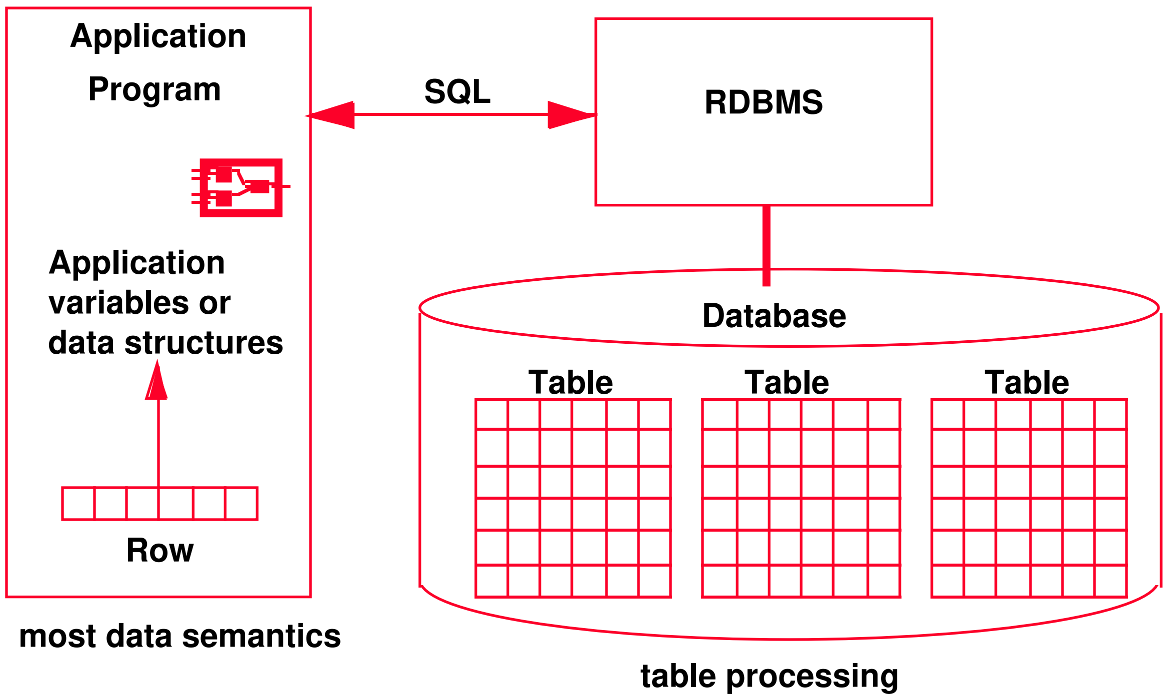


Figure 1.1 Relational DBMS Access

The capabilities of relational DBMSs, together with the pattern of application interaction with them described above, have proved highly effective in dealing with many mainstream business application areas. However, in attempting to apply relational DBMSs to the more advanced applications mentioned earlier, it has become widely recognized that the capabilities of conventional relational DBMSs required extensions to address the additional requirements of these applications.

One area in which extensions are required is in the *data types* supported by DBMSs. The collection of built-in data types in most relational DBMSs (e.g., integer, character string) and built-in operators (e.g., +, -, average) were motivated by the needs of simple business data processing applications. However, in many more advanced applications this collection of types is not appropriate. For example, in a geographic application a user typically wants points, lines, line groups, and polygons as basic data types, and operators which include intersection, distance, and containment. In scientific applications, a user requires complex numbers, arrays, and time series with appropriate operations. Business applications also need specialized data types. [Sto86b] cites as an example a program that computes interest on bonds, and requires all months to have 30 days. Thus, this program would require a data type `DATE` in which the subtraction "April 15" - "March 15" would give 30, rather than 31, days. Other business applications involving even more complex data types have been mentioned earlier. Currently, applications must simulate these data types and operators using the basic data types and operators provided by the DBMS, resulting in the potential for substantial inefficiency and complexity.

For example, consider the simple example of a relation consisting of data on two dimensional boxes [Sto86b]. If each box has an identifier, then it can be represented by the coordinates of two corner points as follows:

```
CREATE TABLE BOX
   (ID NUMBER(4),
    X1 FLOAT(8),
    X2 FLOAT(8),
    Y1 FLOAT(8),
    Y2 FLOAT(8));
```

Now consider a simple query to find all the (non-rotated) boxes that overlap the unit square, i.e., the box with coordinates (0, 1, 0, 1). This might be expressed in SQL as:

```
SELECT *
FROM BOX
WHERE NOT (X2 <= 0 OR X1 >= 1 OR Y2 <= 0 OR Y1 >= 1);
```

There are two basic problems with this representation. First, the query is hard to formulate and understand (the same query for *rotated* boxes would be even more complex). Second, the query would probably run slowly, both because there are numerous clauses that must be checked, and because the query optimizer would probably not be able to optimize the expression very well.

The problem would be considerably simplified if the DBMS provided a `BOX` data type, together with appropriate operations on that data type, allowing the relation to be defined as:

```
CREATE TABLE BOX
   (ID NUMBER(4),
    DESC BOX);
```

and the above query to be expressed as:

```
SELECT *
FROM BOX
WHERE DESC OVERLAPS '0, 1, 0, 1';
```

The definition of this new data type includes defining both a specialized representation for instances of the type, and also specialized operations that apply to instances of the type (e.g., the `OVERLAPS` operator in the example). The need for such types is more crucial as the individual types and their behavior become more complicated (e.g., images), and less expressible (or not expressible at all) using the DBMS's native capabilities.

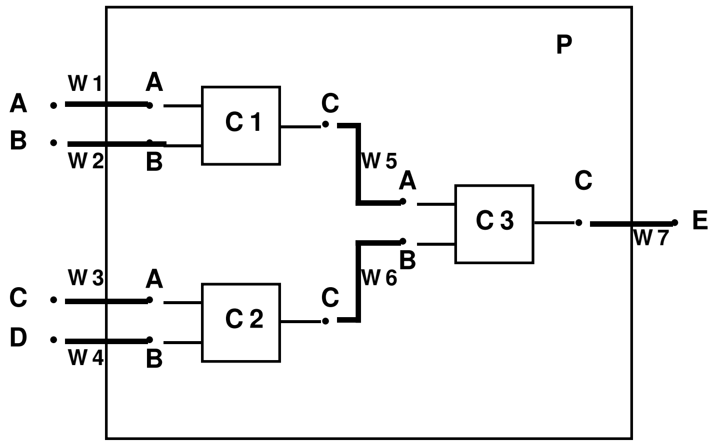
The changes required to support the definition of new types are not restricted to those that can be seen by the user. Internal changes to the DBMS may also be required to support such data types. For example, current database systems implement hashing and B-trees as fast access paths for built-in data types. While some enhanced data types (e.g., date and time) might be able to use existing access methods, given certain extensions, other data types, such as polygons (or the `BOX` type above), would require (or, at least, could take advantage of) entirely new access methods. A number of such access methods appropriate for spatial data have been developed. Specialized access methods for text searching have also been developed, and might be required to provide adequate performance.

One approach that has been taken to address this problem is to define specific extensions for various non-traditional data types, and add them to the DBMS. For example, various special-purpose extensions to DBMSs have been proposed for dealing with text, images, and geometric (e.g., CAD or geographic) data. The difficulty with this approach is that in each case the extensions added are application-specific, and limited in generality. For example, the spatial capabilities required for geographic data would be, at best, of limited use in a CAD application. Moreover, even for a single *category* of data, such as geographic data, there are many different ways to represent and manipulate the data, and each way may be the best in some specific application. It is not really feasible to select one set of data types as built in types, and also provide the required generality. At the same time, it is clearly impossible to build in all useful data types that might be required in any application in the same DBMS. Only a user-defined data type facility allows this type of customization. This would allow the inclusion of customized data types to support the required abstractions, while maintaining generality by allowing additional types to be defined as applications grow or change.

A second area in which extensions are required is in the *structures* supported by DBMSs. Many of the new DBMS applications deal with highly structured objects that are composed of other objects (such objects are frequently referred to as *complex objects* in the DBMS research literature [DMBC+87]). For example, a part in a part hierarchy may be composed of other parts; an integrated circuit module may be composed of other modules, pins and wires; a complex geographic feature such as an industrial park may be composed of other features such as buildings, smokestacks, and gardens; a program module may be composed of other program modules, each with a declaration part and body part; a document may be composed of sections and front matter, and the sections themselves may be composed of section headings, paragraphs of text, and figures.

In many applications these complex objects are the units of storage, retrieval, update, integrity control, concurrency control, and recovery. For instance, in a design application, it may be necessary to lock an entire part assembly (i.e., the part together with all of its component parts) if the part is to be redesigned. Similarly, if an instance of an integrated circuit module is to be deleted from a design, the deletion must be propagated atomically to all its components. The application programs that deal with such complex objects typically represent them as complex graph (pointer-based) structures in main memory, and, when a DBMS is not used, must linearize these structures for storage in conventional file systems.

It has been recognized for some time that such complex objects pose a potential problem for conventional relational DBMSs. This can be illustrated by Figure 1.2, which shows a simple example from [LP83], one of the early papers discussing this problem.



Gate Type	GT	description
	2AND	"C = A&B"
	4AND	"E = A&B&C&D"

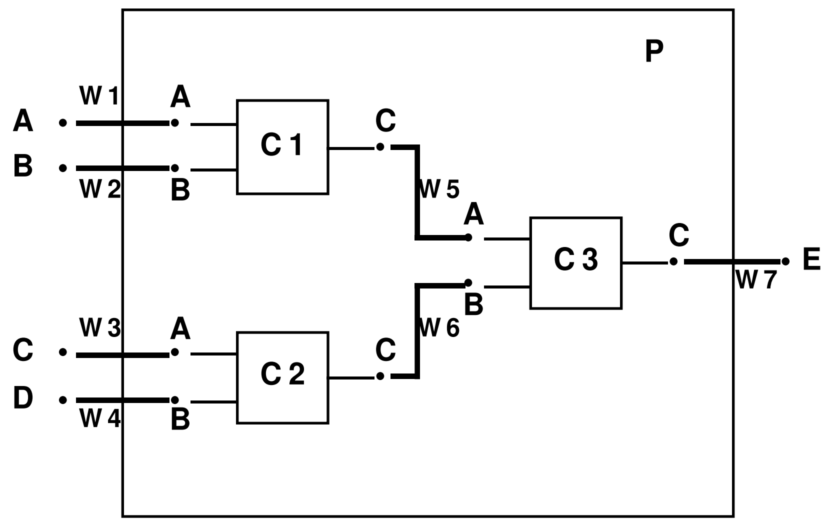
Figure 1.2 Simple Relational Representation of a Complex Object

The figure shows a 4-input AND gate built up of three 2-input AND gates, together with the relation that might be constructed *if no attempt were made to represent detailed information about the gates themselves, or their internal structure, in the database*.

There are two extreme approaches to storing the *details* of such objects in a relational DBMS. The first approach is to completely decompose the object into tuples (rows) so that the entire structure of the object can be directly represented in (and manipulated by) the DBMS. This approach is illustrated in Figure 1.3.

In this case, the `Gate Type` and `Pin Type` relations describe the two types of gates, and the pins for each gate type, respectively. The `Gate Instance` relation indicates that each instance of a 4-input AND gate is built up of three instances of 2-input AND gates. The `Wire Instance` relation shows that each instance of a 4-input AND gate contains seven wires, each connecting a pair of pins. For instance, wire W1 connects the (external) input pin A of the 4-input AND gate to pin A of its first component 2-input AND gate.

There are a number of problems with this approach. As this example illustrates, in order to completely represent the structure of a complex object in a relational DBMS, the object must be broken up into many tuples scattered among several relations. This flattening into tuples of an inherently complex structure may be very unnatural from the user's perspective. Moreover, the manipulation of the object represented in this way may be both complicated and inefficient: First, because there is no way to explicitly specify to the DBMS that all these linked tuples form a single, complex object, operations on the complex object as a whole must typically consist of several relational commands. Second, these relational operations form a transaction that is very difficult to optimize, since the transaction typically involves a large number of joins, and is actually extracting a very few tuples (at most) from each relation, unlike the set-oriented operations for which conventional relational DBMSs are designed. Finally, the application program is again entirely responsible for converting this collection of tuples into the complex graph structure



Gate Type	GT	description
	2AND	"C = A&B"
	4AND	"E = A&B&C&D"

Pin Type	GT	PT	I/O
	2AND	A	I
	2AND	B	I
	2AND	C	O
	4AND	A	I
	4AND	B	I
	4AND	C	I
	4AND	D	I
	4AND	E	O

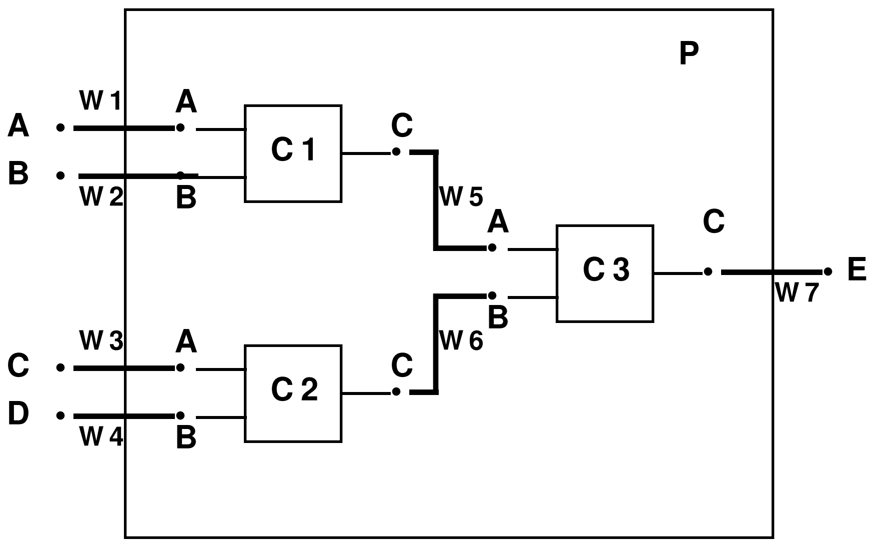
Gate Instance	GT	GI	Parent
	2AND	C1	4AND
	2AND	C2	4AND
	2AND	C3	4AND
	4AND	P	.....

Wire Instance	WI	GT1	GI1	Pin1	GT2	GI2	Pin2	Parent
	W1	4AND	P	A	2AND	C1	A	4AND
	W2	4AND	P	B	2AND	C1	B	4AND
	W3	4AND	P	C	2AND	C2	A	4AND
	W4	4AND	P	D	2AND	C2	B	4AND
	W5	2AND	C1	C	2AND	C3	A	4AND
	W6	2AND	C2	C	2AND	C3	B	4AND
	W7	2AND	C3	C	4AND	P	E	4AND

Figure 1.3 Complete Relational Representation of a Complex Object

that is typically required for internal processing (e.g., for operations that traverse the components of a gate, or follow connections to other gates; performing the database join operations required to simulate such traversals imposes an intolerable performance overhead on realistic applications employing such data types). All this results in considerable performance overhead. Moreover, this example is extremely simple. The situation is, of course, much more complicated when objects of realistic complexity are involved, and database-sized collections of such objects must be managed.

A second approach is to store the entire object as an uninterpreted byte string, often called a *binary* (or *basic*) *large object*, or *BLOB*, in a column of a relation, usually together with some descriptive information in other columns, such as the part number or (here) the object type, that can be used to identify it. This approach is illustrated in Figure 1.4. In the `Gate Type` relation shown in the figure, the values of the `layout` column (depicted in the figure by graphics) would actually be byte strings that, once loaded into an application program's memory space, represent the entire structure of the gate.



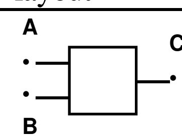
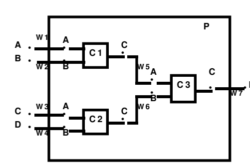
Gate Type	GT	description	layout
	2AND	"C = A&B"	
	4AND	"E = A&B&C&D"	

Figure 1.4 Relational Representation of a Complex Object Using a BLOB

This approach might be perfectly acceptable if all that is required is to display the layout in its entirety (or to display an employee's photograph contained in a similar column in an `EMPLOYEE` relation). However, in this approach the DBMS cannot be used for any manipulation or retrieval involving the content or structure of the object. For example, the subcomponents of the object cannot be identified by the DBMS, since this information is "buried" within the byte string representing the entire object, and cannot be extracted by the DBMS. Moreover, the DBMS is not able to perform any type checking on this data, and may not be able to perform other useful administrative functions. In effect, the DBMS is being used as a (somewhat complex) file system. Finally, if the main-memory form of the object has a complex, pointer-based internal structure (as is true in many advanced applications), the application program must deal with the problem of converting between this structure and a relocatable database structure that can be reasonably represented as a byte string.

Even in fairly conventional business applications, additional structural capabilities would sometimes be desirable. For example, frequently the most natural structure (and the one that might also yield the best performance) for an application would involve a nested or hierarchical structure, such as:

```
Person Name
     First
     Middle
     Last
Person Address
     Street
     City
     State
     ZIP
```

etc., rather than the requirement to normalize or flatten the structure imposed by a relational DBMS. Similarly, sometimes it would be desirable to define general categories of objects, such as class `EMPLOYEE` having attributes like `NAME`, `AGE`, and `SALARY`, together with *subclasses* of those objects, such as `MANAGER`, `SUMMER_STAFF`, `PART_TIME`, `CONTRACT_EMPLOYEE`, etc., with attributes specific to those subclasses, such that the subclasses can be treated as instances of the generic `EMPLOYEE` class in specific cases (such as printing all employees) without having to introduce redundant data or additional complexity in query formulation (e.g., numerous case statements) or processing. The more complex business applications mentioned at the beginning of this section, e.g., those involving document and multimedia types, impose even more stringent requirements, and can involve complex structures similar to those illustrated in Figures 1.2 through 1.4. Examples of these structures are presented in Section 2, and characteristics of ODBMS applications are discussed further in Section 7.

Approximations of the required capabilities can be constructed by users, using combinations of facilities provided by relational DBMSs such as BLOBs and stored procedures, and possibly object-oriented front-ends. However, this is generally recognized as a stop-gap measure. As the next section will show, the overall trend in the industry is to extend the DBMS itself with object-oriented facilities to provide the required capabilities, resulting in the ODBMSs described in Sections 3 and 4. Such facilities are recognized as being the only real way to providing both the required functionality and performance, even in relational DBMSs (as evidenced by the object extensions being developed for the relational SQL standard described in Section 5).

1.2 A Review of the Problem and Solution Approaches

DBMSs exist to provide persistent storage for program data. *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 in the program's address space is transient, i.e., it exists 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 data persist beyond the lifetime of a single program execution, the programmer must include explicit instructions to save the data on stable storage, such as in a file on disk or using a DBMS as an intermediary. Conversely, to reuse data that was created in an earlier execution, or by some other program, the programmer must include explicit instructions to fetch the data from stable storage (or from the DBMS).

The problems illustrated in Section 1.1 are examples of what is sometimes called the *impedance mismatch* between application requirements for handling complex data structures and the DBMS's capabilities to provide persistent support for them. In particular, this impedance mismatch involves differences between:

- application requirements for representing complex data structures (e.g., the complex objects described in Section 1.1) and DBMS capabilities for providing persistent storage for those data structures

- application requirements to access complex data structures in specific ways (e.g., follow pointers in complex graph structures) and DBMS capabilities for supporting those access patterns.

The impedance mismatch between application requirements and conventional DBMS capabilities is exacerbated by the increasing use of object-oriented programming languages, which provide increasing power to define and manipulate complex data structures in the program, without corresponding facilities in the DBMS.

The impedance mismatch problem is being addressed by attempts to provide a more *seamless* interface between programming languages and DBMSs. *Seamlessness* generally describes the situation where the DBMS's data model or type system is a persistent extension of that of one or more host programming languages. There are both ease-of-use and performance issues relating to a lack of seamlessness:

- It is the programmer's responsibility to decide when to save and fetch objects.

- The programmer must write code to translate between the data representations in the program's address space and their external representations on secondary storage (e.g., a file of records or a set of tuples in one or more relations), which may be quite different. This mapping is especially complicated in an object-oriented environment, where a complex object may be composed of many component objects linked together with pointers. The resulting code may make the application harder to understand, and add considerable performance overhead.

- Type-checking also becomes part of the programmer's responsibility. Modern programming languages have elaborate type systems, often with strong type checking. However, when data is written out to stable storage, there may be no type checking provided (if a conventional file system is used) or a different type system may exist (if a conventional DBMS is used). The data can be modified by another program that has access to the file, with no guarantee that the modified object will conform to the original type when it is reread into the address space of the program that created it.

- The capabilities provided by the DBMS to access data may not match the ways in which the application wants to access it. This mismatch may result in poor application performance. This is true even if the DBMS optimizes queries, as noted already.

Conversely, there are significant performance advantages in having a closer correspondence between programming language and persistent (DBMS) types. Basically they involve minimizing the processing required to translate between application program objects and database objects. These have been shown to be crucial in many advanced applications where attempts were made to use a relational DBMS. There are also advantages in program development, maintenance, and reuse. Ideally (depending on the degree of seamlessness obtained), these include:

- Programmers do not have to write the often-complex code required to convert from programming language representations to database representations. The same type system is used in both places, and the system can automatically handle any conversion that is required.

- Programmers can write code that uses persistent storage in the same way in which they write ordinary code, without learning new concepts.

- Code originally written in a non-persistent environment can be imported and used in a database environment without modification (or without significant modification).

In addition, it is increasingly desirable in large-scale database applications for the DBMS to be capable of supporting logically-centralized behavior, such as business rules, and standard procedures associated with certain data types, to ensure that data semantics are maintained and key rules are enforced on all applications. The ability provided by some relational DBMSs to support stored procedures, triggers, and alerters are simple examples of such facilities. However, generalized facilities for specifying user-defined types with their own type-specific behavior, and for specifying behavior in the database in the same way as it is specified in applications, would provide increased support for such large-scale applications.

Previous research on "database programming languages" attempted to address some of these problems by integrating a programming language and a DBMS. These languages allowed certain distinguished programming language data types to be persistent (e.g., the type "relation" in PASCAL/R [Sch77]; "entity types" in ADAPLEX [SFL83]). These languages also incorporated powerful query facilities that allowed the programmer to fetch sets of data objects from the database, in some cases following pointer-based data relationships, in one access. Also, by allowing database operations to be bracketed in transactions, they supported the controlled sharing of concurrently accessed data objects. However, the programmer still saw two type systems (and two address spaces), one for the transient objects in the program's address space, and another for the persistent objects in the database, and was responsible for translating between them.

More recent work on "persistent programming languages" has been predicated on the principle of "orthogonal persistence" [ABDD+87], i.e., *any* instance of any programming language data type should be allowed to persist, not just some distinguished types. This principle has motivated the design of persistent programming languages and object managers for them (e.g., Trellis/Owl [OBS86] and Galileo [AGOO86]). [AP87] contains a very thorough discussion of the issues involved in developing persistent and database programming languages, and a very thorough survey of work done in the area to that point.

Early research to address the need for extensions from the DBMS side frequently involved attempts to directly extend the relational model. For instance, the `BOX` example from [Sto86b] cited at the beginning of Section 1.1 illustrates an approach to adding new data types developed for relational systems (in fact, this specific research, together with additional developments, is incorporated in the Illustra ORDBMS described in Section 4.2). Similarly, the focus of the early work on complex objects was largely to address the problem within the relational model. For example, [HL82, LP83] proposed enhancements to the relational model (and to the SQL language) to represent complex "entities" and hierarchical relationships among them. More recently, the relational model has been extended to more directly represent hierarchical structures through the concept of nested (or non-first-normal-form) relations, i.e., relations whose attribute values may themselves be relations. The relational algebra and calculus have correspondingly been extended to manipulate and retrieve such hierarchically structured objects [SS86, FT83, BK86]. Storage structures and query processing techniques for such relations have also been investigated [DPS86, Kho87]. E. F. Codd, the developer of the original relational model, at one point defined an extended form of the relational model, RM/T [Cod79], which, for example, included support for subclasses as described above.

A somewhat parallel thread of development was the investigation of new data models and associated DBMSs which, while sometimes based on relational techniques or relational principles, had at least the *appearance* of being substantially different from the relational model. Early examples of such models are the Entity-Relationship model [Che76], and the *semantic data models* such as TAXIS [MBW80], SDM [HM78], and Daplex [Shi81].

A more recent development is the *Object-Oriented DBMS* (OODBMS). The OODBMS combines capabilities of conventional DBMSs and object-oriented programming languages such as Smalltalk [GR83]. While some early work on OODBMSs can be considered as an extension of work on semantic data models [Bro81], most work on OODBMSs has involved the merger of database and programming language technology. OODBMSs provide facilities that go considerably beyond most semantic data models, and generally have been based on attempts to directly provide all the capabilities described earlier, such as complex objects, type extensibility, subclassing, etc. OODBMSs have been and continue to be the subject of much research, and a number of commercial OODBMS products have appeared, some of them relatively mature. The early commercial OODBMSs, and the early attempts to generalize relational systems, were the primary subjects of [Man89a].

A new class of DBMS, the *Object/Relational DBMS* (ORDBMS), constitutes the most recent piece of the puzzle. ORDBMSs are to some extent the results of research on extending relational DBMSs, but are also based to some extent on the emergence of OODBMSs as viable products. The ORDBMSs attempt to combine the capabilities of an OODBMS and a relational DBMS (and current individual products represent different mixtures of these capabilities). While many of the OODBMSs originally emphasized their non-relational capabilities, with query facilities "added on" or otherwise considered less than crucial, ORDBMSs are based on the idea of full relational query support as basic, with the addition of objects (or abstract data types) as additional data structure concepts that can be used to generalize relational tables and types used within them. The appearance of ORDBMSs, together with the general support now provided for query facilities (and object extensions to SQL) by OODBMSs, indicates that the merger of OODBMS and relational capabilities described in [Man89a] is taking place, and that even more complete mergers can be expected as OODBMS and ORDBMS (ODBMS) technology matures further. Moreover, the appearance of these two classes of products indicates 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.

This section has described the limitations of relational DBMS technology, and mentioned some of the historical developments, that have led to the current generation of ODBMS products. The next section gives a general overview of the characteristics expected in such products, as an introduction to the description of the actual products in Sections 3 and 4.

2. Characteristics of Object DBMSs

The previous section noted that the appearance of OODBMS and ORDBMS products indicates a general convergence on the *object* concept as a key mechanism for extending DBMS capabilities. Moreover, the general support now provided for query facilities (and object extensions to SQL) by OODBMSs indicates more complete mergers of capabilities of these classes of systems can be expected as OODBMS and ORDBMS (ODBMS) technology matures further. This section gives a general overview of the characteristics expected in such products, as an introduction to the description of the actual products in Sections 3 and 4.

Since both classes of products generally agree on the need to incorporate object concepts, this section begins with a review of basic object concepts, and then goes on to describe the general characteristics that are desirable in a DBMS based on these concepts.

2.1 Basic Object Concepts

The original notion of objects is generally attributed to the Simula programming language [DN66]. However, object-oriented programming did not emerge as a new programming paradigm until the development of the Smalltalk language [GR83]. At the present time, many object-oriented languages have been developed, and some are in wide use. The following paragraphs, taken from [Man89a] present a brief overview of the object concept as it is often conventionally understood. There has been a great deal of debate about what facilities a language or system must have in order to make it "object-oriented," and object languages differ greatly, even in some of the fundamentals. As a result, the presentation here primarily uses the concepts and terminology of Smalltalk, which is considered a fairly "pure" object-oriented language. Useful surveys of object-oriented language principles, and the variants that exist among the various languages, can be found in, e.g., [SB86, Weg90, YT87]. In addition, collections of papers that illustrate the scope of object-oriented technology in general can be found in the various OOPSLA Conference proceedings, e.g., [Mey86].

Most familiar programming languages use an *operator/operand* model of computation, which involves the use of separate procedures and data. In the operator/operand model, a user (programmer) is given a set of data (operands), and a set of operators (functions or procedures) which take various combinations of operands as parameters. The user is responsible for determining which operations to apply to any data to be manipulated, and for making sure that the operands match the type requirements specified for the operations being called. Languages employing this model include C, Pascal, and FORTRAN. If the same type of operator (say `print`) must be performed on operands of different types, separate procedures must be defined and used (e.g., `prnPoint(point)` for data of type `point`, or `prnRectangle(rect)` for data of type `rectangle`). This is because an operator name typically identifies a distinct operator or procedure (piece of code), and the actual implementations of the operators will probably differ for different types.

Some languages, such as Simula, Ada, and CLU [Lis81], allow an extension of this approach in which the programmer specifies an abstract operation name, and the compiler differentiates which specific routine to call at compile time, based on the type of the operand. For example, the programmer might write `print(point)` or `print(rect)`, but the compiler would refer to a different procedure in these cases.

In the typical object-oriented language or system, this *operator/operand* model is replaced by an *object/message* model. In the object/message model, all information is represented in the form of *objects*. An object is a self-contained entity which consists of:

- Its own private memory or *state*, typically consisting of a set of slots or *instance variables* (Smalltalk) which can hold values or references to other objects.

- A set of operations (sometimes called its *protocol*). These operations constitute the object's external *interface* with the rest of the system

All interaction with an object occurs through requests (called *messages* in Smalltalk) conceptually sent to the object requesting it to execute one of the operations in its interface. A computation is defined as a sequence of interactions among objects in the system. Message sending is a form of indirect procedure call. Instead of naming a procedure to perform an operation on an object, the object, called the *receiver*, is identified and a message sent to it. A *selector* (operation name) in the message specifies the specific operation to be performed. Parameters to be passed to the operation are also specified in the message. For example, in Smalltalk the programmer might write the messages `aPoint print` and `aRect print` to print the point object `aPoint` and the rectangle object `aRect` respectively. In Smalltalk, the receiver is specified first, followed by the selector. In this case, the messages have no parameters[1].

An object that receives a message is responsible (conceptually, at run time) for deciding how to respond to the message, using its own private procedures (called *methods*) for performing the requested operation. These methods have the exclusive capability of accessing and manipulating the object's state (this is referred to as *encapsulation*). A method is like a short subroutine that can perform computation itself, and can also send messages to other objects (known to the method via references in the object's state) to ask them to perform various operations (and return appropriate results).

In the typical object-oriented system, objects are defined as members of one or more classes or types[2]. The class defines the internal form of every object of the class, as well as defining the protocol associated with objects of the class. In many object-oriented languages, an object class is itself represented by a *class object*. As noted above, the abstraction provided by an object-oriented language is that each object is a self-contained entity having its own operations. However, in a class-based language, the class object stores, among other things, the code for the methods of all objects of the class, so they need not be duplicated in each individual object.

A user can freely define new object classes, which can then be used in the same way as object classes supplied with the language. This allows the language to be *extended* with classes specific to application requirements. New object classes (*subclasses*) can also be defined as extensions of existing ones by a technique called *inheritance*. The existing class

---

[1] Many object languages use a notation that resembles an ordinary subroutine call, rather than the Smalltalk message notation. In such languages, the message to print the point would be written `print(aPoint)`, but have the same meaning as the Smalltalk message. In these languages, the first argument to the call typically designates the receiver, with the other arguments being parameters. An exception to this rule is found in the *generalized object models*, in which *all* the arguments are equally "receivers" of the message. Generalized object models are described further in Section 2.10.

[2] The distinction between "class" and "type", where there is one, is irrelevant for purposes of this report. The Smalltalk term "class" will generally be used for the rest of this section.

is then referred to as the new class's *superclass*. If class `Employee` is specified as inheriting from class `Person`, this means that objects of class `Employee` will respond to the same messages as objects of class `Person`, in addition to the messages defined specifically for class `Employee`. A given superclass `Person` may have many subclasses (e.g., `Employee`, `Student`, `Retiree`), with each subclass inheriting from `Person`. (In some object-oriented systems, a given subclass may have several superclasses (e.g., `Teaching_Assistant` may have superclasses `Student` and `Employee`), inheriting selected messages from each superclass. This is referred to as *multiple inheritance*.) Moreover, a given subclass may itself be the superclass for still other subclasses, and so on for many levels. This structure of classes and subclasses is referred to as a *class hierarchy* or *lattice*.

The use of inheritance in object definitions assists in defining object protocols in a relatively structured way. This, coupled with the use of messages, provides a form of *polymorphism*. Polymorphism is the capability of the same message to elicit a different (but semantically similar) response depending on the class of the receiving object. This facility enables a program to treat objects from different classes in a uniform manner. Thus, in the example above, a program could send a `print` message to a heterogeneous collection of `point` and `rectangle` objects, and each object would "print itself" using the method for printing that was defined for that class of object.

Object-oriented languages have generally been associated with specific implementation techniques, such as the dynamic binding of messages to method code, and automatic storage management, in the form of garbage collection. For example, in Smalltalk, methods are not called directly by name, but indirectly at run time via a *dispatch table* associated with the object class. Each object contains a pointer to its class object. When a message is sent to a particular object, the class pointer is followed, and an attempt is made to find a corresponding method in the class object. If the method is not found there, the object's superclass in the hierarchy is searched, and so on upward until the root class of all objects (class `Object`) is found. If the method is not found there, an error is reported. This process of finding the proper method based on the selector (operation name) and the class of the receiving object is referred to as *dispatching*. Run-time dispatching provides a great deal of the flexibility generally associated with object-oriented programming, but adds additional overhead (although, when various compiler optimizations are used, the overhead is often minimal). In other object-oriented languages, such as C++ [Str86], a more static, compile-time binding of operation name to method is used, trading some flexibility for performance.

In object-oriented languages, each object has a system-defined unique identifier (called an *object-oriented pointer*, or *oop*, in Smalltalk, and more frequently referred to as an *object identifier*, or *oid*). Relationships between one object and another are represented directly, by storing the oid of the second object in an instance variable of the first, and in a pure object-oriented language like Smalltalk, values such as strings are almost entirely replaced by object pointers. *Garbage collection* is felt to be important in object-oriented languages such as Smalltalk in order to guarantee the safety of object pointers due to the highly dynamic nature of object creation and deletion. In these languages, there is no way to explicitly delete an object. Rather, objects are garbage-collected when they are no longer referenced by any other object. Again, however, other object-oriented languages, such as C++, prefer to trade some safety for improved performance and use explicit object deletion operations rather than garbage collection.

The general characteristics of objects that can be defined in a given object-oriented language, such as whether classes exist or not, what kinds of inheritance exist, or whether explicit deletion is allowed or not, are often referred to as that language's *object model* [1].

Figure 2.1.1 shows some simple object classes and objects that might be defined for a Computer-Aided Design (CAD) application, and that illustrate various capabilities of object-oriented systems. The top-left object represents a `Part` object class. The object class has `display`, `weight`, `subParts`, and `part_number` methods. The top-right object represents a `Part` object (instance), with the associated messages that can be sent to it shown around the outside, and instance variables representing the state of the individual object shown within the object. A `weight` message sent to this object would return a number (object) representing the weight of the part. A `subParts` message sent to the object would return the set of `Part` objects that are subparts of this part. The exact (implementation) mechanism used by the object to respond to these messages is encapsulated within the object, and is not visible outside the object.

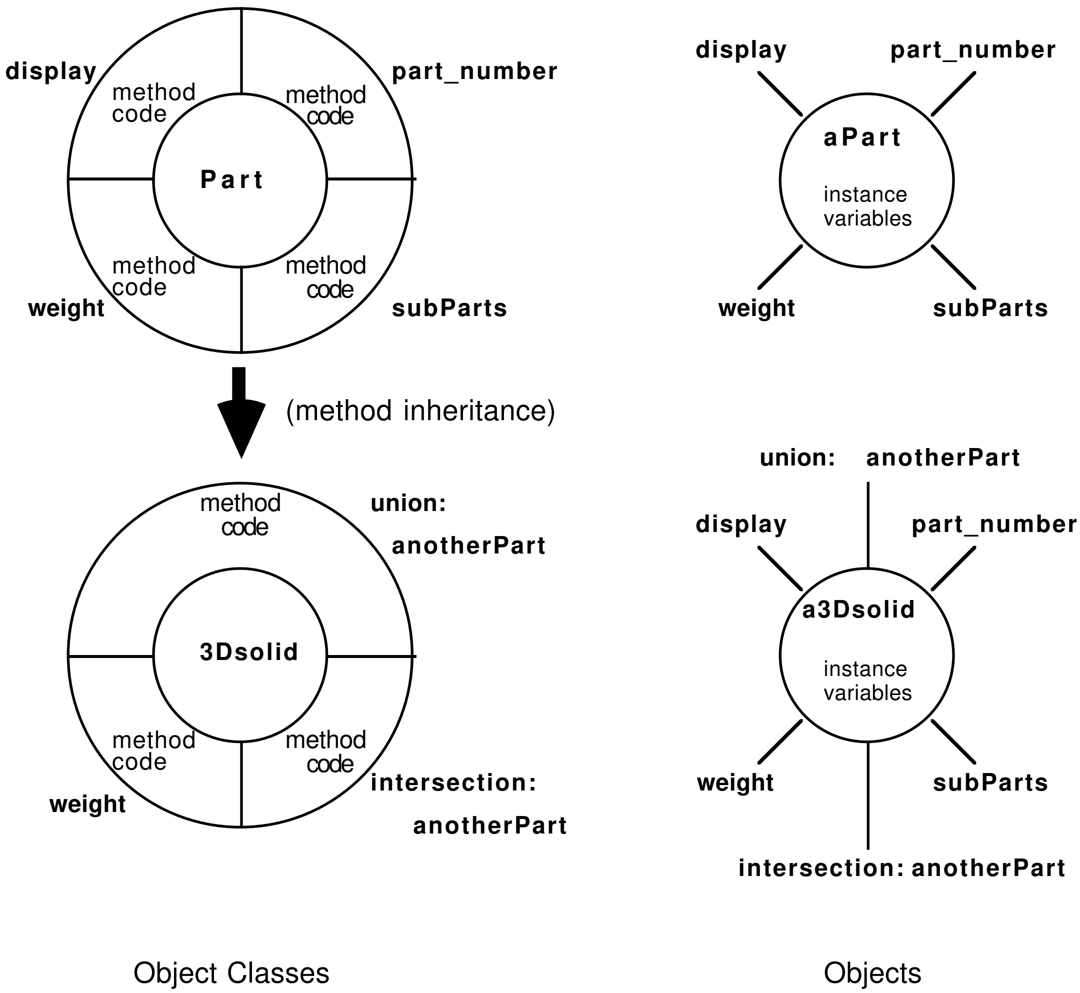


Figure 2.1.1 Object Classes and Objects in a CAD Application.

[1] The concept of an *object model* corresponds roughly to the concept of a *type system* in a conventional programming language and to the concept of a *data model* in a conventional database system. Further discussion of object model concepts may be found in [Mai89, MZ89, Man89b].

The bottom-left object represents the object class for a specialized type of `Part` object, a `3Dsolid` object. Because it is defined as a specialization of object class `Part`, the `3Dsolid` object (instance) on the bottom right is shown as inheriting all the messages that apply to `Part` objects. In addition, the `3Dsolid` object class defines two additional methods that apply only to `3Dsolid` objects, `union` and `intersection`. `union` and `intersection` messages sent to a `3Dsolid` object include another `3Dsolid` object as an argument, and return the `3Dsolid` that represents the union or intersection, respectively, of the argument object and the original object. The `3Dsolid` object class also *redefines* the `weight` method inherited from class `Part`. For example, the `weight` method for class `Part` might look up the value of the part's weight in an instance variable, while the `weight` method for class `3Dsolid` might compute the weight from information stored in instance variables about the part's geometry and the density of the part's material. Because the implementation used to respond to the `weight` message is encapsulated within the object, other objects requiring weight information can send `weight` messages to parts without regard to what type of part they are dealing with, or exactly how the weight information is derived for a particular part. Due to the inheritance of methods from class `Part`, only the three new methods must be explicitly defined for object class `3Dsolid`.

A number of advantages are generally cited for object-oriented programming:

- Defining a system in terms of objects facilitates the construction of software components that closely parallel the application domain, thus assisting in the design and understandability of systems.

- The use of objects and messages encourages modular design, since (at least in a pure object-oriented language) the implementation of one object cannot depend on internals of another, only on how it responds to messages. In addition, because the private information of an object can only be accessed by the methods of the object, modularity is reinforced, and software can be made more reliable.

- The use of classes and inheritance provides a simple and expressive model for the relationship of various parts of the system's definition, and also assists in making components reusable or extensible in the construction of revised or new systems.

2.2 Object DBMSs

Object database management systems (ODBMSs) [BM93, Bro91, Cat94b, Loc85, DD86, Dit88, KA90, Kho93, KL89, ZM89] generally attempt to capture the same concepts as object-oriented programming, but also provide the additional characteristics necessary to support large, shared, persistent object stores. These characteristics include support for set-oriented requests (queries), efficient processing over large secondary storage organizations, concurrency control, and recovery. These systems have been developed to address the problems posed by advanced database applications, as described in Section 1.

The general idea is that an ODBMS is a DBMS that stores not data, but *objects* in the sense described in the last section: encapsulated combinations of data structures together with associated procedures (methods). This allows instances of arbitrary kinds of data to be stored within the "database", as shown in Figure 2.1.2. The ODBMS provides data integration, overall control, and DBMS support facilities for all types of objects. Applications using any of these types can then communicate via the shared database.

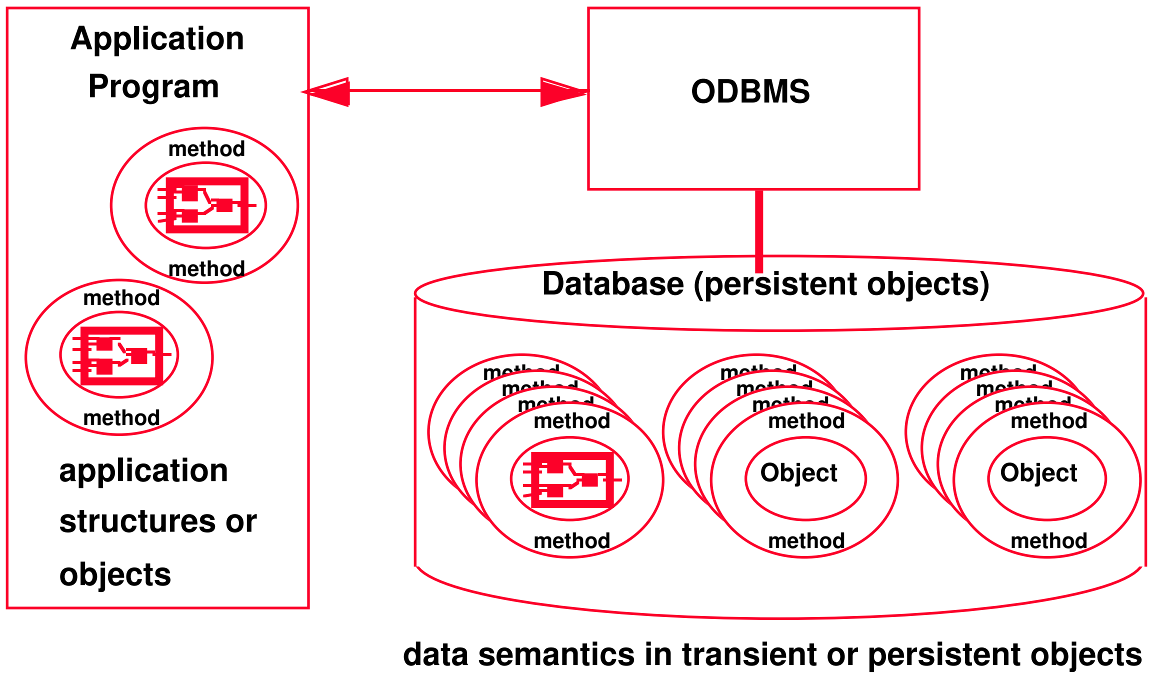


Figure 2.1.2 An Object DBMS.

Compared to a conventional relational DBMS, a typical ODBMS generally differs in at least the following respects:

- An ODBMS supports *user-defined* data structures and operations on them (object classes and methods), rather than restricting users to instances of a single built-in type (relations) and a fixed set of operations on them (relational operations).

- An ODBMS supports the concept of *object identity*. This means that the object has an identity independent of its attribute values (this identity can be used to reference the object from within other objects).

- An ODBMS supports *direct object relationships*. This means that objects related to a given object can be accessed by invoking a method of the given object (such as the `subParts` method in Figure 2.1), which generally looks up the object identifiers of the related objects in instance variables. Thus, related objects can be treated as if they were part of the given object's internal data, rather than always requiring a relational *join* operation on attributes of the object (although ODBMSs often also support value-based join operations).

ODBMSs generally also support such concepts as inheritance and, due to the advanced applications for which they are intended, also frequently support such things as advanced concurrency control (object sharing) mechanisms, mechanisms to maintain multiple versions of objects, and extensive predefined object class libraries.

The stored procedures and BLOBs often found in relational DBMSs are not truly comparable to the facilities provided by an ODBMS for supporting user-defined types and operations. For example, [Loo94] notes that BLOBs and stored procedures are not really enough to make an ODBMS, but are just the raw material from which one could construct primitive object-like facilities. Specifically, with stored procedures and BLOBs, it is possible to define code that will operate on data of arbitrary forms, but there is really no way to define combinations of specific groups of procedures together with specific data formats as being new *types* that are both recognized in languages (programming languages, query languages, schema declarations) and enforced during language translation and at run time. Similarly, with stored procedures and BLOBs there is no coherent coupling of specific procedures with specific units of state to form objects. Finally, stored procedures and BLOBs provide no support for inheritance.

The basic idea in using an ODBMS is to represent an entity or object in the application domain being modeled with a corresponding object in the database. This includes modeling the *behavior* of each object as well as the object's structure and relationships to other defined objects. This one-to-one mapping reduces the "semantic gap" or impedance mismatch between the application domain and the database modeling of that domain. Moreover, when coupled with object-oriented application programs, this reduces the impedance mismatch between these programs and their supporting database data.

An ODBMS generally comes with the following facilities (many of which are familiar from relational DBMSs):

- An object class or type definition facility (a DDL).

- A built-in programming language and/or one or more interfaces to conventional programming languages. These are used not only as conventional application programming interfaces (APIs), but also for defining the object methods (behavior) that form part of class/type definitions.

- A query language facility (usually, but not always, some form of SQL extended in various ways to deal with objects).

- An object manager or database "engine" that handles the basic DBMS operations.

- An object class library of predefined object classes used in making the ODBMS functionality available to programs, and to facilitate the writing of object methods.

- A set of tools. These include database administrator facilities, such as schema design tools, graphical database interfaces, performance tuning utilities, etc.

There is currently no standard object model or standard language(s) for ODBMSs corresponding to the status of the relational model and SQL for relational DBMSs. The Object Database Management Group (ODMG), an industry consortium of OODBMS vendors, has proposed a standard for ODBMSs (see Section 5.1). In addition, ANSI and ISO standards committees are currently developing object extensions to SQL, the standard relational language [MS93]. The extended language draft is generally referred to as "SQL3" [Mel94] (see Section 5.2). An approved SQL standard containing object facilities will ultimately play a primary role in defining the concept of an ORDBMS. The possibility of merging these two standards, or of ODBMSs that support both, is also discussed in Section 5.

These standards will ultimately define the key characteristics of the ODBMSs that conform to them. Even now, current products sometimes have at least some characteristics defined by these standards, to the extent that current product characteristics have contributed to these standards, or products have been developed anticipating certain facilities being in the standards. In addition, however, two earlier database "manifestos" have been published which have attempted to define the characteristics that must be satisfied by next-generation database systems, specifically "object-oriented database systems" (OODBMSs) [ABDD+89] and "third generation database systems"[1] [CADF90]. These sets of characteristics have attained wide circulation in the database field. In the rest of this section, we summarize the extended DBMS requirements identified in these manifestos, in order to provide a general overview of the desired characteristics for ODBMSs (another set of ODBMS requirements, partly based on those of [ABDD+89, CADF90], can be found in [Cat94b]).

[ABDD+89] notes that, based simply on the term *object-oriented DBMS*, an OODBMS must satisfy two generic criteria: (a) it must be an *object-oriented* system; (b) it must be a *DBMS*. By similar reasoning, criteria (a) and (b) also apply to ORDBMSs (even though [ABDD+89] does not discuss a separate category of "ORDBMSs"), together with the additional criterion that: (c) an ORDBMS must be *relational* (which for the moment can be taken to mean that it provides at least the capabilities of a relational DBMS). In [ABDD+89], criterion (a) translates into eight mandatory features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility, and computational completeness. Criterion (b) translates into five mandatory features: persistence, secondary storage management, concurrency, recovery, and an ad hoc query facility. However, for expository purposes these features are grouped differently than in the manifestos. Specifically, we first cover the capabilities identified in [ABDD+89], indicating coverage of related capabilities identified in [CADF90] (in some cases, these related capabilities from [CADF90] extend, or even disagree with, those included in [ABDD+89]). As in [ABDD+89], we divide these capabilities into *mandatory*, *optional*, and *open* capabilities. However, we include in our mandatory capabilities those cited as "mandatory" in *either* manifesto. Finally, section 2.2.11 provides additional discussion of what it means for an ODBMS to be an ORDBMS, i.e., criterion (c). This is necessary since both manifestos were formulated prior to the existence of the current ORDBMS products (hence neither covered this question explicitly), and also because what it means for an object-oriented system to also be relational (or vice-versa) is trickier to define that it may appear.

In each case, we illustrate the capabilities using multimedia data type examples from [MHB90]. Multimedia types are used to represent the general sorts of extended applications to be supported by ODBMSs, since these types include complex documents, images, and other types likely to be used in such applications. As noted earlier, we use the abbreviation *ODBMS* (Object DBMS) to refer generically to both OODBMSs and ORDBMSs. For the most part, the two manifestos are relatively consistent in their requirements, and the text will make clear when they differ. Moreover, for the most part

---

[1] [CADF90] essentially describes requirements for extended relational DBMSs, and does not consider object facilities per se. However, as the following sections will demonstrate, the extensions suggested by [CADF90] include user-defined abstract data types, encapsulation, etc., all of which could be supported by adding objects to a relational DBMS. As a result, [CADF90] can, in some respects, be taken as defining requirements for ORDBMSs.

the requirements contained in the manifestos are incorporated in the specifications of the two ODBMS standards cited above, as well as in the capabilities of the ODBMS products described in Sections 3 and 4.

2.2.1 Complex Objects and Object Identity

An ODBMS must enable users to construct objects with *complex state*, i.e., that are comprised of or reference other objects (as well as values, e.g., integers or strings) in order to model application objects, such as documents, with potentially-complex internal structures. They also need to model complex interrelationships among defined objects to correctly represent application semantics. In addition, an ODBMS should provide *constructors* for building complex objects, e.g., sets and tuples, from simpler ones in order to marshall arguments and move results. [ABDD+89] notes that such constructors should be applicable to objects of any type, including other aggregates, to produce nested structures.

Complex objects are objects that are built from simpler ones by applying *constructors* to them. The simplest types of objects (sometimes called *atomic types*--those that have no subcomponents) are types such as integers, characters, and Booleans. Typical complex object constructors include tuples, sets, bags (sets which may contain duplicate elements), lists, and arrays. [ABDD+89] identifies *sets*, *lists*, and *tuples* as the minimal set of constructors an ODBMS should have. [CADF90] identifies *sets*, *sequences* (lists), *records*, and *arrays* as desirable constructors, and also indicates that a *union* type constructor would be desirable (i.e., the ability to define the value of an attribute as coming from one of a set of types. In addition, [CADF90] indicates that it would be desirable to support *functions* as a built-in type (this is related to the capabilities of *computational completeness* and language *closure* described later). Both manifestos indicate the importance of being able to recursively compose the constructors. That is, it should be possible to apply any constructor to form any type of object (e.g., the ability to form a list of arrays, an array of lists, etc.).

These constructors allow complex objects to be built because they allow a constructed object to contain references to other objects. Complex objects are frequently found in multimedia applications. An example of such a complex object is the Document object shown in Figure 2.2.1.

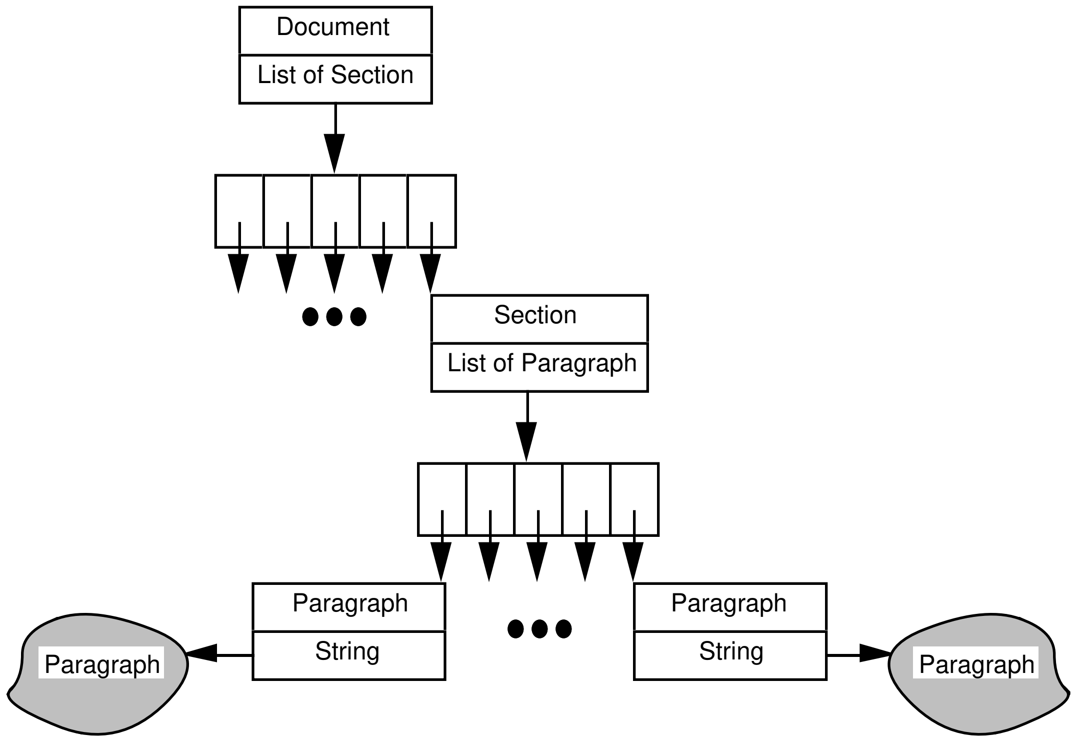


Figure 2.2.1 A Document (Complex) Object

The corresponding (somewhat oversimplified) definitions from [MHB90] are given below.

```
Create Type Document
Supertypes: TextObject
Properties:
      title: String;
      author: String;
      components: List of Section;
Operations:
      print;
      checkSpelling: String {errors};
      checkGrammar: String {errors};

Create Type Section
Supertypes: TextObject
Properties:
      title: String;
      number: Integer;
      components: List of Paragraph;
Operations:
      print;
      checkSpelling: String {errors};
      checkGrammar: String {errors};

Create Type Paragraph
Supertypes: TextObject
```

```
Properties:
      paragraphText: TextString;
Operations:
      print;
      checkSpelling: String {errors};
      checkGrammar: String {errors};
      getSentence (index: Integer): TextString;
      changeFont (String);
```

An object is said to have *identity* if there is a way to reference it independently of its current state [MZ89], as opposed to a tuple in a relational database system, which must be accessed via a key value that is part of the tuple state. Generally, object identity is implemented by means of *object identifiers* (*oids*) assigned explicitly by the system (*object-oriented pointers* (*oops*) or *surrogates* are terms often used synonymously). [ABDD+89] considers support for object identity a requirement for an OODBMS. On the other hand, proposition 1.4 of [CADF90] differs from this position in stating "unique Identifiers (UIDs) for records should be assigned by the DBMS only if a user-defined primary key is not available". This supports the use as identifiers of primary keys as they are currently used in relational DBMSs. Provided that appropriate identifier uniqueness and referential integrity constraints are maintained, and the user-defined primary keys are not subject to frequent update, the two approaches can be considered structurally equivalent.

A primary use of identifiers is in building complex objects, where they serve to identify the various components of the complex object. For example, in the definition of type Document above, a list of identifiers (of objects of type Section) is used to represent the sections of a document. Identifiers are of particular use in expressing the concept of *shared structure*. This is because they allow a distinction to be made between two objects having as subcomponents two different objects that are alike (have common state and behavior) and two objects having the *same object* as a subcomponent.

Both the ODMG and SQL3 specifications currently support variants of both complex values (having no object identifiers) and complex objects (having identifiers), using many of the constructors identified above.

2.2.2 Extensibility Using Abstract Types or Classes

[ABDD+89] requires that an OODBMS must support either *types* or *classes*. There are OODBMSs in both camps. Briefly, a *type* describes the common features of a set of objects with the same characteristics. This is the same as the programming language concept of type. In many type-based systems, types are used primarily at compile time, and are not first-class objects (e.g., they cannot be accessed or modified at run time). A *class* is similar to a type, but has more of a run time connotation. In most class-based systems, the class is itself an object, which contains facilities for constructing new objects of the class (e.g., by sending the class object a `new` message), as well as an *extension*: the set of all instances (objects) of the class in the database. There are strong similarities between the type and class concepts, and the differences in some systems can be very subtle.

Support for one of these concepts is necessary because it should be possible in an ODBMS to define a specification to which some collection of objects is defined to correspond, since the system must support situations in which there are many "similar" objects. This is a standard capability in DBMSs. However, [ABDD+89] does not require that an OODBMS automatically maintain an extension for every type.

A conventional database system includes a set of predefined types (e.g., integer, character). [ABDD+89] requires that this set of types be *extensible*, in the sense that there is a way for a user to define new types, and there is then no distinction in usage between system-defined and user-defined types. [CADF90] agrees with the desirability of supporting "abstract data type facilities for constructing new base types".

These *abstract data type* facilities involve the concept of *encapsulation* of object state (data) and behavior (operations), as found in object-oriented systems. The idea is based on programming language abstract data types, where an object has an *interface* part, which is made available to users of the object, and an *implementation* part, which is hidden from them. In an object-oriented system, clients using the object can access the object only via its defined external interface [Sny86]. This provides a form of logical data independence which allows changes to the implementation of an object (the representation of its state or the programs that implement its operations) without the need to change programs or queries using the object. Encapsulation both hides unnecessary details of the implementation and ensures modularity. [ABDD+89] requires object encapsulation for an OODBMS, and [CADF90] considers encapsulation "a good idea". The need for this capability is particularly important in situations where there may be many different representations for objects that might need to be considered as instances of the same type by users, or in situations where types may be defined and implemented by independent groups or outside vendors, and imported into a system. In such situations, it is very helpful not to be dependent on detailed information about object internals in order to use them.

Multimedia data types such as images and sounds are examples of such types. This can be illustrated by reference to the type `Image` described below.

```
Create Type Image
Supertypes: Object
Properties:
      name: String;
      owner: Person;
      creation: Date;
      lastModified: Date;
      encodingType: String;
      recordingAlgorithm: String;
      rawData: BLOB;
      description: Text;
      pixelWidth: Integer;
      pixelHeight: Integer;
      pixelDepth: Integer;
      colorMap: ColorMap;
      compressionAlgorithm: String;
Operations:
      display (VideoOutputDevice);
      compress (ratio: Real): Image;
      extractSubimage (subimage: Region): Image;
      overlay (Image): Image
      reverse (subimage: Region): Image
      dither (subimage: Region): Image
      magnify (factor: Integer): Image
      reduce (factor: Integer): Image
      match (Image): Boolean;
```

The `match` operation determines if an image I1 is "similar" (using some definition of similarity) to another image I2[1]. This obviously could be quite a complex procedure, and it would be unfortunate if, in order to use this operation, a user had to be familiar with the details of the representation of the image (which might differ from one image to another), and the details of the algorithm used. This would be concealed by the encapsulation.

[ABDD+89] defines *computational completeness* in programming languages as the ability to express any computable function. It notes that while computational completeness is rare in the DMLs of conventional database systems (in particular, standard SQL is currently not computationally complete), in OODBMSs it is usually provided by a reasonable connection to an existing programming language. Computational completeness is a requirement because it must be possible to define the behavioral aspects (the operations or *methods*) of the objects in the system when specifying user-defined data types. These operations may be arbitrarily complex (too complex to be expressed in a conventional database query language such as SQL), as illustrated by the operations defined for type `Image` above. This requirement is intended to rule out a system that provides for state representation only in terms of, for example, instance (state) variables, and provides only access and assignment operations for those instance variables. However, it does not require that the operations be defined only in terms of an entirely new database language. Procedures written in arbitrary programming languages and linked to the objects in some way would be acceptable. [ABDD+89] requires this capability. [CADF90] indicates that "functions, including database procedures and methods...are a good idea". It also indicates that third-generation DBMSs must be accessible from multiple higher-level languages. SQL3 provides extensions to make SQL a computationally-complete programming language, in part to address the need to define object operations in types defined using SQL3's abstract data type extensions. Section 5.2 discusses this further.

2.2.3 Class or Type Hierarchies

[ABDD+89] requires an OODBMS to support *inheritance*, which allows an object to borrow some of its state or behavior from another object. Specification of inheritance involves forming class or type *hierarchies* that organize specifications of object interfaces and implementations, making them more concise, by factoring out the shared parts. For example, given that type Person has been defined, with properties such as name and age, it would then be possible to define a *subtype* Employee of type Person which shared those properties by specifying only the additional properties (such as salary and employer) peculiar to Employee objects. [ABDD+89] identifies several forms of hierarchy, i.e., of *subtyping* or *sub-classing* that differ in what is factored out (behavior specifications, constraints, etc.), but does not prescribe a particular form. [CADF90] indicates that inheritance "is a good idea", and indicates a particular interest in what [ABDD+89] defines as *constraint* inheritance, an example of which would be the definition of a teenager as being a subtype of person with an age between 13 and 19.

Multiple inheritance is a generalization of inheritance which allows an object to inherit properties or operations along more than one path in the class or type hierarchy. For example, if both Employee and Student are defined as subtypes of type Person, one might wish to define Student-Employee objects representing persons that are both students and

---

[1] The syntax for sending a message in the hypothetical object model being used here is similar to that of Smalltalk, in which the object to receive a message is syntactically distinguished from any arguments that may be supplied. So the message to match image I1 against image I2 would be `I1 match (I2)`.

employees, and that inherit properties from both types. [CADF90] considers multiple inheritance "essential", while [ABDD+89] considers it optional.

[ABDD+89] notes that inheritance helps code reusability because every program is at the level at which the largest number of objects can share it. Type hierarchies are particularly important in defining the complex types used for multimedia data, both in allowing the reuse of what might be relatively complex code, and also in allowing the organization of the related types. An example of such a (single inheritance) hierarchy is the hierarchy of graphics objects described in [MHB90]. The relationships among these types are shown in Figure 2.2.2, while several of the type definitions are given in Figure 2.2.3. The type `GraphicsObject` specifies that each graphical object has a `startPoint` and a `color`. Specific graphical objects are subtypes of `GraphicsObject`. Some of the operations defined for `GraphicsObject`, although applicable to all instances of its subtypes, must be redefined by those subtypes since the supertype `GraphicsObject` does not have direct access to its subtype's properties.

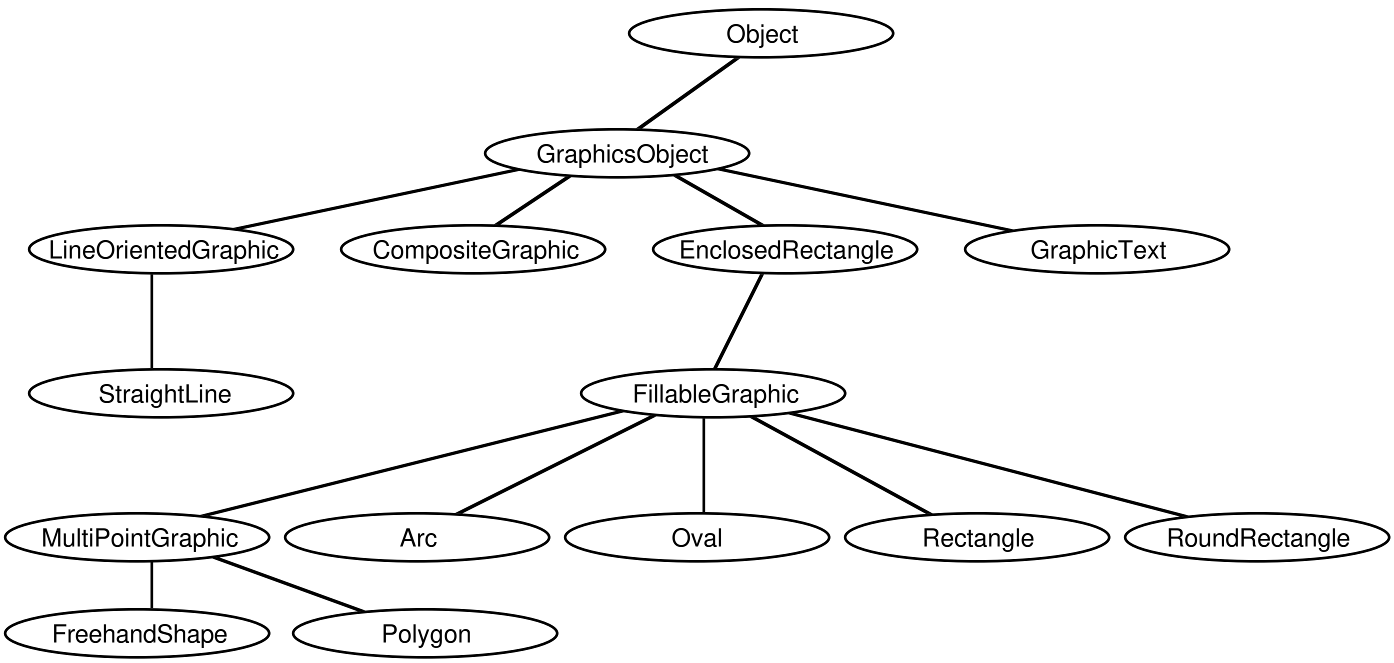


Figure 2.2.2 Graphics Type Lattice

```
Create Type GraphicsObject
Supertypes: Object
Properties:
      startPoint: Point;
      color: String (Blue|Red|Magenta|Green|Cyan|Yellow|Black|White);
Operations:
      draw (Window);<Redefine in subtype>
      scale (percent: Integer);<Redefine in subtype>
      duplicate: GraphicsObject;
      rotate (degrees: Integer);<Redefine in subtype>
      flipHorizontal;<Redefine in subtype>
      flipVertical;<Redefine in subtype>
      moveToFront;
      moveToBack;
      moveForward;
      moveBackward;
      overlaps (GraphicsObject): Boolean;

Create Type CompositeGraphic
Supertypes: GraphicsObject
Properties:
      components: List of GraphicsObject;
Operations:
      draw (window1: Window)
            {
            for i=1 to components.numElements
                  components[i].draw (window1);
            }
      group (List of GraphicsObjects): CompositeGraphic;
      ungroup: List of GraphicsObjects;

Create Type EnclosedRectangle
Supertypes: GraphicsObject
Properties:
      relativeEndPt: Point;
Operations:

Create Type LineOrientedGraphic
Supertypes: GraphicsObject
Properties:
      thickness: Integer;
      linePattern: Bitmap;
      dashedLine: Bitmap;
Operations:

Create Type FillableGraphic
Supertypes: EnclosedRectangle
Properties:
      fillPattern: Bitmap
Operations:
```

Figure 2.2.3 Example Definitions of Graphic Object Types

Multiple inheritance can be illustrated by a portion of the `PhysicalDevice` type lattice from [MHB90], shown in Figure 2.2.4. Such types are useful at lower levels of system architectures to encapsulate procedures that interface with the actual hardware. Device types can be used to represent metadata about the device (e.g., serial numbers and purchase prices that might be useful for administrative purposes) as well as the attributes of the device required by the system. In Figure 2.2.4, type `Telephone` is defined as inheriting from both types `AudioOutputDevice` and `AudioInputDevice` (multiple inheritance), since it has characteristics of both types of devices (this hierarchy is for purposes of illustration only; there is no claim that this is necessarily a reasonable use of inheritance under the circumstances). Of course, in addition to its inherited characteristics, the telephone has additional characteristics which are defined by the properties and operations of the type `Telephone`.

The declarations in Figure 2.2.5 illustrate a problem that can occur with multiple inheritance, *name ambiguity*. In this case, type `Telephone` inherits properties named `upperfreqresp`, `lowerfreqresp`, and `impedance` from *both* types `AudioInputDevice` and `AudioOutputDevice`. Thus, a reference to the property `upperfreqresp` of type `Telephone` would be ambiguous; it would not be clear which of the two inherited properties with the same name was being referred to. The approach used to deal with this problem in the declarations below is one of the standard approaches, *renaming* the inherited properties in the subtype so that they have unique names. Note that, because no renaming is required for the `sensitivity` property inherited from type `AudioOutputDevice`, this property is not mentioned in the declarations for type `Telephone`.

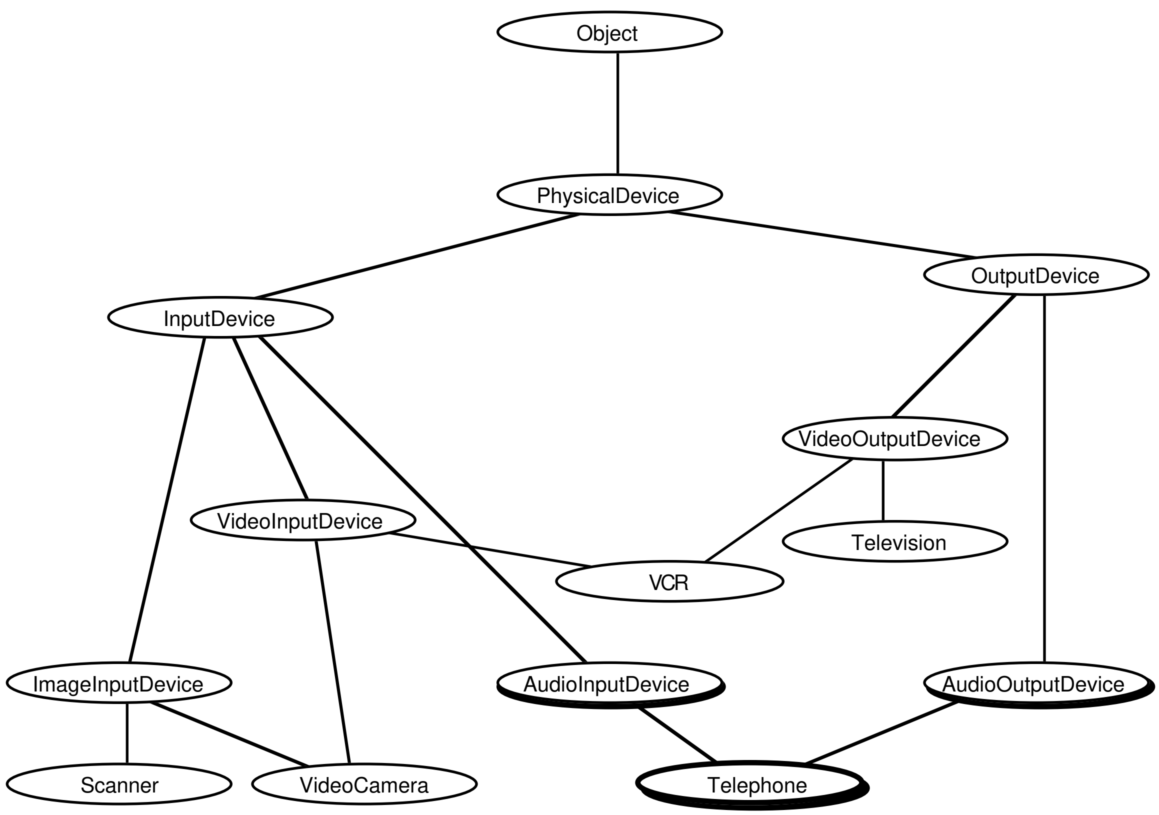


Figure 2.2.4 Portion of `PhysicalDevice` Type Lattice

```
Create Type PhysicalDevice
Supertypes: Object
Properties:
      make: String;
      model: String;
      serialNumber: String;
      purchasePrice: Dollars;
      purchaseDate: Date;
Operations:

Create Type AudioInputDevice
Supertypes: InputDevice
Properties:
      upperfreqresp: Hertzvalue;
      lowerfreqresp: Hertzvalue;
      impedance: Integer;
Operations:

Create Type AudioOutputDevice
Supertypes: OutputDevice
Properties:
      upperfreqresp: Hertzvalue;
      lowerfreqresp: Hertzvalue;
      impedance: Integer;
      sensitivity: dB;
Operations:

Create Type Telephone
Supertypes: AudioOutputDevice, AudioInputDevice
Properties:
      inputupperfreqresp: AudioInputDevice.upperfreqresp;
      inputlowerfreqresp: AudioInputDevice.lowerfreqresp;
      inputimpedance: AudioInputDevice.impedance;
      outputupperfreqresp: AudioOutputDevice.upperfreqresp;
      outputlowerfreqresp: AudioOutputDevice.lowerfreqresp;
      outputimpedance: AudioOutputDevice.impedance;
Operations:
      ring;
      redial;
      hold;
      answer;
      dial (phoneNumber: String);
```

Figure 2.2.5 Example Definitions of `PhysicalDevice` Object Types

2.2.4 Overriding, Overloading, and Late Binding

[ABDD+89] requires that an OODBMS support some form of operation name *overloading* and operation *overriding* (this is also sometimes referred to as *polymorphism*), which allow the same name to be used for different operations (depending on the type or class of the object to which the operation is applied) and the implementation to be redefined in the type or class, so that the same name denotes different programs. These capabilities allow invocations of the same operation *name* to result in the execution of different operation *code* based on the type of the object to which the operation is applied, and is a standard feature in object-oriented systems. For example, these capabilities would allow a user to apply a display operation to a set of graphics objects or images of different types, without concern for the fact that the different objects may implement that operation in different ways (for example, the display operation for a picture would probably differ from the display operation for a character string). This sometimes involves the run-time (*late*) binding of operation names to operation code.

This is illustrated by the type `CompositeGraphic` described in the last section. Instances of `CompositeGraphic` contain a list of `GraphicsObject` instances which may include other `CompositeGraphic` instances. The operation

```
myCar.draw (window1);
```

where `myCar` is an instance of type `CompositeGraphic`, would invoke the `draw` operation defined for `CompositeGraphic`. This operation is defined, as shown below, to invoke the `draw` operation for each component of the `CompositeGraphic` instance, including other contained instances of `CompositeGraphic`.

```
draw (window1:Window)
     {
     for i=1 to components.numElements
          components[i].draw (window1);
     }
```

Thus the invocation would be propagated down the hierarchy of component objects until all primitive graphic objects have been drawn. This definition works because all the graphical objects, composite or not, use the same (overloaded) operation name `draw` for drawing, even though the code invoked might differ depending on whether the object in question is composite or primitive, and on the particular shape of the object.

2.2.5 Persistence and Secondary Storage Management

A *persistent* object is one that continues to exist beyond the end of the process that created it, such as one residing in a file or database. Support for persistent objects is an obvious requirement for an ODBMS to truly be a database system, and hence [ABDD+89] requires this capability. [ABDD+89] also goes further to state that persistence should ideally be *orthogonal*, which means that an object of any type should be allowed to become persistent as such (without explicit translation), and *implicit*, which means that the user should not have to explicitly move or copy data to make an object persistent. [CADF90] relates persistence to the DBMS's interface with multiple languages, by stating that "persistent X for a variety of X's is a good idea (where X denotes a programming language). They will all be supported on top of a single DBMS by compiler extensions and a (more or less) complex run time system".

[ABDD+89] also requires an OODBMS to manage very large databases, and interprets this to mean that an OODBMS must provide secondary storage management mechanisms such as index management, clustering, buffering, access path selection, and query optimization. [ABDD+89] goes on to emphasize that these should be invisible to the user (e.g., an application programmer should not have to write code to maintain indices, or to move data between disk and main memory). [ABDD+89] states this requirement largely in system terms, and thus it is not clear whether an object model containing an explicit cluster construct (such as is found in a number of current OODBMSs) would be necessarily ruled out by the requirement for invisibility. [CADF90] explicitly addresses this point by stating that (proposition 2.4) "performance indicators have almost nothing to do with data models and must not appear in them".

These capabilities will be important in supporting databases containing multimedia information, both in application units such as documents, and also in repositories of miscellaneous multimedia information that can be used for a variety of purposes (examples in the PC world are the "clip art" repositories containing collections of scanned images). Advanced secondary storage management capabilities are particularly important in efficiently performing operations on multimedia data types such as sounds, images, and large units of text, since the instances are frequently quite large, and operating on them (e.g., text searches) can require considerable data movement.

Neither the object *model* for an object-oriented system nor a language based on such a model would necessarily explicitly contain a description of secondary storage management mechanisms such as index management, clustering, buffering, access path selection, and query optimization, any more than the relational model per se contains descriptions of such mechanisms. (The specification of a given system based on such a model would, of course, contain such descriptions.) [CADF90]'s proposition 2.4, however, is a model or language constraint, in that it prohibits explicit constructs for clustering or other performance-related details, taking the view that these should be specified in lower-level facilities explicitly intended for performance-related specifications.

2.2.6 Concurrency and Recovery

[ABDD+89] requires that an OODBMS support transaction processing and concurrency control mechanisms offering the same level of service as provided by current database systems. This is necessary so that concurrent user requests can be processed in a safe but efficient manner. [ABDD+89] goes on to indicate that serializability of operations should at least be offered, although less strict alternatives may also be offered. [ABDD+89] also requires that an OODBMS support recovery mechanisms offering the same level of service as provided by current database systems. These recovery mechanisms typically interact with the facilities provided for concurrency control.

Transaction processing, concurrency control, and recovery mechanisms are of interest in connection with multimedia and other advanced data types not only because ODBMSs will need to support transactions involving these types, but also because these data types sometimes impose distinct requirements on these mechanisms. [ABDD+89] notes that, in many of the new applications for which OODBMSs are intended, the transaction model used in conventional DBMSs is not satisfactory. In particular, transactions tend to be very long, and involve cooperating applications, rather than conflicting applications, which means that the usual serializability criterion is inappropriate. [ABDD+89] refers to these as *design transactions*, and lists support for them as an optional feature for an OODBMS. However, they may very well be required in multimedia and other extended applications.

One of the goals of current research is to allow ODBMSs to provide more sophisticated concurrency and recovery mechanisms that support the more complex requirements of multimedia and other advanced applications, and that take advantage of opportunities provided by certain user-defined data types to improve concurrency. A simple example cited in [Ahm91] is that of a user-defined type `Queue` which defines operations `Enqueue()` and `Dequeue()`. If Q is an instance of type `Queue`, then `Enqueue(Q,X)` places an element X on the input end of the queue, and `Dequeue(Q,Y)` removes an element Y from the output end of the queue. `Enqueue()` and `Dequeue()` would be viewed by a traditional concurrency control mechanism as write operations, and would require transactions to acquire write locks on Q to execute either. Thus, if one transaction issues an `Enqueue()` and another issues a `Dequeue()` on Q, one must wait for the other to complete. However, as long as the queue is non-empty, it is actually possible for the two transactions to proceed simultaneously without interference. Taking the semantics of the types and operations into account allows the mechanism to achieve greater concurrency than a traditional approach would allow. Research toward such extended transaction mechanisms for distributed object systems and ODBMSs includes work by the DOM project at GTE Laboratories on a general environment for specifying and implementing extended transaction models [GH92, HG93].

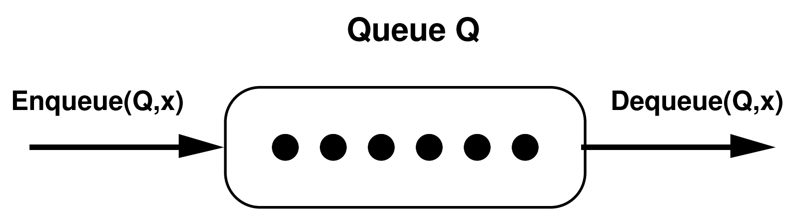


Figure 2.2.6 Operations on a Queue

2.2.7 Ad Hoc Query Facility

[ABDD+89] requires that an OODBMS support object-oriented facilities corresponding to the facilities provided by the query languages found in relational database systems. [CADF90] identifies similar requirements, which can be taken to apply to ORDBMSs. Queries in relational database languages may be viewed as *select* operations on the aggregate type *relation* (or table) [Weg87]. They have the general form:

select(relation,predicate)

Query complexity and efficiency are determined by the nature of the predicate. Relational query languages specify all queries in terms of a restricted set of relational query primitives (the *relational algebra*, or *relational calculi* defined in terms of the algebra) whose optimization has been extensively studied. Query languages for ODBMSs must accommodate the greater richness of ODBMS type systems (e.g., the incorporation of user-defined types and inheritance) for which optimization is not as well understood. For example, the predicate in an object query would involve testing the state of various objects by invoking operations on the objects. The exact nature of these operations might be encapsulated within the object, and thus unknown to the optimizer, making the task of optimization extremely difficult. This issue is discussed further in Section 6.

[ABDD+89] does not require that query facilities be provided in the form of a query language per se: they might also be provided in a graphics-based facility, or as part of a data manipulation language. However, [ABDD+89] does indicate that a query facility should be: (a) *high-level* (largely declarative), (b) *efficient* (the interface lending itself to some form of query optimization), and (c) *application-independent* (capable of being applied to any possible database). Also, in spite of the lack of a requirement for a query language in [ABDD+89], most OODBMSs provide a query facility based on some variant of SQL, and all of the ORDBMSs provide such facilities. Moreover, while SQL3 is by definition "SQL-like", so is the OQL query language in the proposed ODMG standard (see Section 5.1).

[CADF90] identifies a number of additional requirements related to a third-generation DBMS query facility:

- "Essentially all programmatic access to a database should be through a non-procedural, high-level access language" (proposition 2.1).
- "There should be at least two ways to specify collections, one using enumeration of members and one using the query language to specify membership" (proposition 2.2).
- "For better or worse, SQL is intergalactic dataspeak" (proposition 3.3).
- "Queries and their resulting answers should be the lowest level of communication between a client and a server" (proposition 2.4).

In an interesting overview of the subject of object query language requirements, [BTA90] summarized a number of features of object query languages, in an attempt to distinguish those that are commonly available from those that are more unusual. Those identified as "common" among object query languages are listed below. [BTA90] went on to suggest that these features, due to their common availability, could be used as the basis of standards work on object query languages.

- Select-From-Where paradigm from SQL;
- Path expressions (for traversing object relationships);
- Support for inheritance;
- Explicit joins;
- Set-valued attributes (including set comparison operators, and sometimes quantifiers);
- Nested queries (dynamic set construction using query expressions, together with set-valued attributes);
- Aggregate functions (e.g., Count, Average);

On the other hand, [BTA90] listed the following features as "controversial", meaning that they are supported by only a few existing object query languages, and that further research and development is needed before there is an acceptable general consensus regarding the feature.

- Data abstraction (restricting the query language to the object interface, as opposed to allowing direct access to state);

- Functions with side effects (several object query languages allow only functions without side effects in queries), and language support for operation specification;

- Queries resulting in derived relationships between objects (e.g., queries that return tuples of oids of related objects) and derived behavior;

- Null values;

- Recursion;

- Relationship to programming language type system (e.g., is there one object query language that is embedded in various programming languages, as with conventional SQL, or is there an OQL[X], tailored for each language);

- Orthogonal treatment of types and type extensions (i.e., dealing with objects that are not in any set, including type extensions);

- Seamlessness in integrating the query language with programming languages, and in dealing with persistent and transient objects;

- Strong typing;

- Result generation and presentation (i.e., dealing with the problem of what is to be presented to users as query results when object queries return only object identifiers);

Both the ODMG query language (called OQL) and SQL3 support all of the "common" facilities, and most of the "controversial" capabilities, identified above. Section 5 discusses these languages further.

2.2.8 Capabilities Not Agreed on as Mandatory or Optional

There were a number of facilities for which the authors of [ABDD+89] could not reach a consensus as to whether they should be mandatory or optional for an OODBMS. These features are listed below. With the exception of views, for which additional research is necessary, these facilities are generally considered requirements for a general-purpose DBMS, and are generally provided (to a greater or lesser extent) in existing ODBMS products.

a. <u>views and derived data</u>

In a relational DBMS, a *view* facility allows the definition of relations that are not explicitly stored, but are computed from stored (or *base*) relations. An extension to this idea allows for arbitrary data (such as individual fields) to be derived from stored data. It would be useful for an ODBMS to support object-oriented facilities corresponding to these, for deriving new objects from existing ones. For example, a view mechanism would allow objects to selectively reveal different aspects of behavior, or provide visibility of different collections of objects, depending on the view. [CADF90] specifically considers updatable views as essential (proposition 2.3).

Construction of views and derived data poses particular requirements on the database language. In particular, the language must be capable of dynamically constructing new data types and their instances. This requires that the language have certain *closure* properties. In addition, for true completeness it must be possible to construct object *operations*, not just object *structures*, in derived data and views. This requires that the language allow operation objects (or instances of the *function* built-in type described in Section 2.2.1) to be both created as the result of a set of operations, and then "applied" as operations.

b. <u>database administration utilities</u>

It is important for an OODBMS to provide a complete set of database administration utilities. Examples include utilities for reorganizing the database, monitoring statistics, keeping audit trails, and archiving. The object *model* or language of an object-oriented system does not explicitly specify such utilities. However, such utilities would necessarily be based indirectly on the object model, since the object model would constrain the implementation details of the system, and would themselves require language facilities to use them.

c. <u>integrity constraints</u>

Integrity constraints, which are statements that must always be true for database contents, can be used by the DBMS to help ensure the correctness and consistency of the database. Some relational DBMSs provide a *trigger* facility to support such constraints. Triggers are actions that can be initiated on access to particular data items, and can be used either to check that constraints hold, or to perform additional operations to bring the database to a consistent state. In an ODBMS, some forms of constraints may be included in the definition of object operations. However, in other cases it may be more appropriate to allow the definition of constraints independently of individual object operations, as in the case of triggers. [CADF90] explicitly indicates that "rules (triggers, constraints) will become a major feature in future systems. They should not be associated with a specific function or collection" (proposition 1.5).

The use of rules as a way of defining behavior required for multimedia objects is described in [MHB90]. Such rules may be defined independently of specific operations or objects (or collections of objects). However, rules may also be explicitly associated with specific objects, in order to improve performance.

Neither [ABDD+89] nor [CADF90] explicitly mention security facilities, although it may be that they were intended to fall under database administration utilities or integrity constraints. In any case, view and rule facilities can also provide the basis for object-based security capabilities.

d. <u>schema evolution</u>

Schema evolution in an ODBMS refers to the problem of modifying type definitions or the type hierarchy (the schema) of a database while preserving consistency between the changed definitions and the existing object instances in the database. Ideally, it should be possible to modify the schema in arbitrary ways without shutting the system down. However, in practice this can be very difficult to do for some types of schema modifications. This becomes a particular issue in the context of the extensible systems being described here, because the creation of new types or type relationships may be a more frequent activity in these systems.

2.2.9 Optional Capabilities

In addition to the mandatory capabilities listed above, [ABDD+89] also identified a set of capabilities which, while they clearly would *improve* an OODBMS, were not required to define the system as *being* an OODBMS. Two of these capabilities, multiple inheritance and design transactions, have been discussed already. The remaining optional capabilities are listed below:

1. Type Checking and Type Inferencing

ODBMSs differ in the amount of type checking and type inferencing (the ability to automatically determine the type of derived data) they can do. This depends critically on the definition of the type system, and is not prescribed in [ABDD+89]. A great deal of research on database languages has to do with issues relating to type checking and inferencing. For example, one of the motivations for the development of database programming languages has to do with an interest in preserving strong typing across the interface between the programming language and the database language environments. Type considerations are also important in the development of optimizable and closed query languages. These issues are a more important concern in ODBMSs than in conventional relational ones because more and different types are being defined, and more complex structures produced by database operations.

2. Distribution

Support for distributed objects is considered optional in [ABDD+89] because a system's support for distribution is independent of whether the system qualifies as an OODBMS. Similar considerations apply to [CADF90]. However, in the general types of computing environments in which advanced applications are likely to be developed, it is essential to provide access to objects residing on distributed components. This involves the need for associated database languages to provide whatever additional facilities are necessary for specifying the characteristics of objects distributed across a computer network. Most ODBMSs are implemented using client/server architectures, and some are implemented as fully distributed architectures (see the descriptions of the individual systems in Sections 3 and 4). Moreover, the integration of OODBMSs with the CORBA distributed object architecture defined by the Object Management Group is explicitly considered in the ODMG standard described in Section 5.1.

3. Versioning

Many of the applications for which OODBMSs are intended involve various forms of "design" activity, and thus require support for keeping track of the different *versions* of a design (or components of a design) that may be developed or under development at the same time. Many of the OODBMSs described in Section 3 have built-in support for versioning concepts. However, [ABDD+89] does not consider this a core requirement. Multimedia and other extended applications might well make heavy use of versioning facilities, e.g., in preparing different versions of a multimedia presentation tailored for different audiences, or for different presentation environments (using different devices). This would require language facilities to, for example, selectively access different versions of the same object, or alternatively to refer generically to the current version (or all versions) of the object.

2.10 Open Choices

In [ABDD+89], the *open choices* are considered to be those areas where there is a degree of freedom for an OODBMS designer. These differ from the mandatory capabilities in the sense that no consensus has been reached by the database community about them. They also differ from the optional capabilities in that it is not yet clear which of the alternatives is more or less object-oriented. These open choices are listed below:

1. Programming paradigm

[ABDD+89] does not insist that an OODBMS choose any particular programming paradigm, such as logic programming, functional programming, or imperative programming. Any of these (or a collection of them) would be considered a legitimate model for an OODBMS. The conventional paradigm is *imperative*. However, in multimedia applications there may be more direct involvement by non-programmer users in the development of multimedia objects. Thus, higher-level scripting languages (possibly graphical) involving dataflow, rule-based and other paradigms may be of increasing importance in these applications. [MHB90] described the use of some alternative paradigms in describing dynamic multimedia object requirements.

As a language-related point, [ABDD+89] also notes that "...the choice of the syntax is also free and people will argue forever whether one should write `john hire` or `john.hire` or `hire john` or `hire(john)`." (However, there is some dependency between this syntax and the type system, as noted in the type system discussion below).

2. Representation system

[ABDD+89] notes that the representation system is defined by the set of atomic types and the set of constructors. Section 2.2.1 described some possible atomic types (elementary types from programming languages) and constructors (tuples, sets, lists) that might be used to define the representation of objects. However, these types and constructors could obviously be extended in various ways. Of particular importance in multimedia and other advanced applications is representation support for BLOBs (variable length byte strings) and their associated operations.

3. Type system

The only type formation facility explicitly required by [ABDD+89] is encapsulation (i.e., an object constructor for defining new objects). Other type formers could also be included in an ODBMS, such as *type constructors* (e.g., `set(T)` where T is an arbitrary type) or union types. Both [ABDD+89] and [CADF90] identify various useful type constructors, as discussed in Section 2.2.1. Type constructors are particularly important in supporting database language requirements. As noted earlier, the ODBMSs described in Sections 3 and 4 generally support a range of such constructors.

One aspect of the type system which can have some significance is whether the object model is *generalized* or *classical*. Classical models are, in many ways, subsets of generalized object models but they are founded on different metaphors [FKMT91].

A *generalized object model* does not distinguish a recipient from other request parameters. In generalized models, a request is defined as an event which identifies an operation and optional parameters. For example, an `AddToInventory(part1, lot2, bin3)` request does not give special significance to either the part, the lot, or the bin. These objects participate uniformly in the request. Sometimes, this requires giving the method implementing the operation access to the internal representation of all types participating in the request. However, a generalized model allows for the easy definition of methods which may be specialized for different combinations of the subtypes of the request parameters. For example, the operation `display(object, device)` might have many different implementations, each specialized for a particular combination of `object` and `device` subtype.

A *classical object model* (sometimes called a *messaging* object model) does distinguish a recipient. In classical models, a request is defined as an event which identifies and operation, a recipient, and optional parameters. Either the part, the lot, or the bin could be designed to be the recipient of the `AddToInventory` request. The request to a specific recipient is called a *message*. A common syntax places the recipient first: `part1.AddToInventory(lot2, bin3).`

Most object-oriented programming languages implement classical object models, examples being Smalltalk and C++. The Common Lisp Object System (CLOS) [Ste90] is an example of a programming language implementing a generalized object model. [ABDD+89] does not prescribe either type of object model, and ODBMSs exist that support either type of model. For example, the OODBMSs discussed in Section 3 generally support classical object models. However, the OpenODB ORDBMS discussed in Section 4 supports a generalized object model, and the SQL3 specification discussed in Section 5 does also.

4. <u>Uniformity</u>

This has to do with how uniform the various constructs in the ODBMS system or model are (or appear to be to users). A key issue of uniformity is whether types and operations are treated uniformly as objects, or whether these are considered as distinct things. Different decisions may be made at the implementation level (e.g., is type information stored explicitly as objects), and at the object model and language level (e.g., are types first class entities in the semantics of the model). [ABDD+89] does not specify any particular approach to uniformity.

2.2.11 Object/Relational Capabilities

Section 1 noted that the Object/Relational DBMSs attempt to combine the capabilities of an OODBMS and a relational DBMS. Many of the OODBMSs originally emphasized their *non*-relational capabilities, particularly their seamless interfaces with object-oriented programming languages, and often considered query facilities as "add ons" or otherwise less than crucial. On the other hand, ORDBMSs generally start with the idea of providing full relational query support, and add objects (or abstract data types) as additional data structure concepts that can be used to generalize relational tables and the types used within them. ORDBMSs (so far) have generally not emphasized seamless programming language interfaces. This general description, while it may convey something of the "flavor" of the products, does not really give a very precise characterization of what distinguishes an ORDBMS from an OODBMS.

In fact, such a characterization is currently rather elusive, and a matter of considerable debate. At the moment, there are no generally-accepted criteria defining what it means 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.

Considering only the externally-visible capabilities provided by the DBMS, the definition of what it means to be "relational" is fairly well established. The relational model is fairly precisely defined, and associated concepts such as the *relational completeness* of database languages are also well defined. Similarly, although there is still a lot of debate on the subject, what it means to be "object-oriented" is generally understood. Certainly the requirements cited in [ABDD+89] are reasonably understandable, even though there will always be debate about whether one object model is preferable to another, as well as debate over other details. However, what it means to *combine* the concepts of "relational" and "object-oriented" is not well-defined. Certainly it is not generally agreed that simply adding the two sets of ideas together can be done without compromising the current definitions on one side or the other. For example, [MD93], a discussion paper addressing issues in the SQL3 definition, notes that "the relational model absolutely depends on...no 'hidden data'. That is, no information other than at row/column intersections", and goes on to note that some ways of combining object facilities with relational ones compromise that principle. Similarly, the extension of such concepts as relational completeness to some extended concept of object/relational completeness has yet to be done in a fully satisfactory way.

As a practical matter, for relational proponents the definition of "object/relational" capabilities will almost certainly be based on the object extensions to the SQL standard, once these are approved. That is, conformance to SQL3 (or some subsequent version of SQL) will be a requirement for any system claiming that it is "object/relational". However, the near-term arguments do not have this standard to go on (since it is currently in a state of considerable flux, as described in Section 5.2). Also, it is not necessarily clear that the OODBMS community will feel that SQL3 is sufficiently "object-oriented", or that the merger of relational and object capabilities has been adequately done to allow the "object-oriented" label to be legitimately used. At the same time, the ODMG OODBMS "standard" (described in Section 5.1) has only recently appeared. While it might be possible to demonstrate some variant of relational completeness or some other conformance by this standard (or some subset of it) to "relational" capabilities, it is not necessarily clear that this would be accepted by the relational community as being adequately relational, particularly if the language syntax involved differed from SQL.

Also, it is not clear that support for a given language (e.g., SQL3), or *any* particular set of external capabilities will actually be satisfactory (except possibly for marketing purposes) in defining an ORDBMS. For example, many ORDBMS proponents argue that just implementing an object SQL extension (or even SQL3) on top of a conventional OODBMS would not make the system truly "object/relational". Instead, they feel that the DBMS should support a relational query engine complete with, e.g., a cost-based query optimizer, in order to be enough like a relational DBMS to be truly "object/relational". This echoes the argument made by relational proponents when relational DBMSs first began to replace DBMSs built using hierarchical or network data models: that it took more than a simple relational query layer to have a relational DBMS; it had to be relational "from the ground up." Lacking this type of argument, it might be possible to argue, for example, that Objectivity/DB (see Section 3.6) is "object/relational", on the grounds that it is object-oriented, but also provides full ANSI SQL support. At the same time, of course, the OODBMS community could well make similar arguments from the opposite direction, i.e., if it is only an object layer on top of a relational "engine", it is not sufficiently "object-oriented".

In the near term, the distinction between OODBMS and ORDBMS will continue to be muddy. In fact, the distinction could easily get muddier, as the OODBMS products move closer to providing the ODMG standard capabilities (whose query facilities are quite SQL-like in many respects), the ORDBMS products move closer to providing object facilities that can be more closely integrated with object programming languages, and both types of products provide more underlying implementation support for the others' capabilities. On the other hand, other possible (and, in fact, likely) developments are that products will begin to provide interfaces for both ODMG- and SQL3-like interfaces, or that both standards will move towards each other. Claims have already been made, for example, that SQL3 could be supported as a concrete syntax of the ODMG OQL query language specifications. The relationships of these standards to each other will be discussed further in Section 5.

As noted in Section 1, the appearance of ORDBMSs, together with the general support now provided for query facilities (and object extensions to SQL) by OODBMSs, indicates that the merger of OODBMS and relational capabilities described in [Man89a] is taking place, and that even more complete mergers can be expected as OODBMS and ORDBMS (ODBMS) technology matures further. Moreover, the appearance of these two classes of products indicates 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.

2.2.12 Summary

In this section, we have summarized the capabilities required for ODBMSs to handle advanced application requirements. We have done this by examining the capabilities identified in two database "manifestos" [ABDD+89, CADF90] describing the characteristics required of next-generation database systems, and illustrating them with examples from [MHB90] used to define multimedia data type requirements. We have also attempted to characterize the difference between an OODBMS and an ORDBMS, and explain why this distinction is as difficult to make as it is. Sections 3 and 4 describe the characteristics of existing ODBMS products of both classes. As the product descriptions in these sections will illustrate, most of the capabilities identified by these "manifestos" are supported by these products.

3. Object-Oriented DBMSs

This section describes (in varying levels of detail) a number of Object-Oriented DBMSs (OODBMSs), as described in the last section. A separate subsection is provided for each system covered. Except for some of the systems described in Section 3.9, the systems covered are all commercial products. Moreover, these systems are specifically OODBMSs, as distinct from the Object/Relational DBMSs (ORDBMSs), which are covered in Section 4 (although, as noted earlier, this distinction is not necessarily easy to make). Some of these OODBMSs are targeted toward the specific database requirements of engineering design applications (and other advanced applications such as computer-assisted publishing), in which relational products might perform badly. These products are designed to function particularly well in these environments, and would not necessarily perform well in the applications for which large relational DBMSs have been developed. Others are intended for more general-purpose environments.

Before proceeding, a few caveats are in order. The intent of describing the various systems covered in this section (and in Section 4) is to given an overall flavor of the characteristics of ODBMSs products, the various facilities that the products provide, and the range of products available. The system descriptions used in these sections have been taken largely from the cited vendor manuals or other published descriptions, and in some cases from discussions with vendor representatives or Internet discussions of the products. While we have attempted to be as accurate as possible in describing the various systems, the descriptions have not been verified by hands-on testing, or submitted to the various vendors for their comments. Also, due to various constraints, the documentation used was of varying degrees of currency, and in some cases the available documentation was rather limited. As a result (and also because new features are introduced in these products with great speed and regularity), the features described are subject to change, or may be unavailable in the current release, or unavailable on certain platforms.

This report is not intended to be exhaustive in describing ODBMSs (either OODBMSs or ORDBMSs), either in terms of describing all possible systems, or all the features of those systems. Thus:

- the omission of some particular system should not be taken as meaning that the system is not worthy of consideration for some specific application
- the lengths of the descriptions should not necessarily be taken as an indication of our opinion of the various systems
- the fact that some feature is not mentioned in the description of a given system should not necessarily be taken as indicating that the feature is not available

This section also does not attempt to provide a detailed comparison of the systems covered against a common set of features or other criteria (Section 6 discusses a number of aspects of ODBMS implementations, and describes the different ways some of the systems implement these aspects, but there is no attempt to provide a general comparison). A very detailed comparison of ODBMSs (including most of those covered in this report) according to an extensive set of specific features (using tables rather than narrative text) can be found in [Bar94a], which is a $1595 2-volume commercial report on the subject (further information can be obtained from DBMSfacts@aol.com).

Readers interested in the selection of a specific product should, as always, evaluate the products in more depth, and using more criteria, than has been possible in this report. In particular, the products should be evaluated with respect to both the specific *applications* to which they will be applied, and the specific *environments* in which they will be applied, since different features will be more or less relevant in different application contexts.

3.1 GemStone

GemStone [BMOP+89, BOS91, OFAQ93, Ric92] is an object-oriented DBMS developed by Servio Corporation. First introduced in 1987, GemStone is the oldest commercial ODBMS available today. GemStone supports multiple external languages, including Smalltalk-80, Smalltalk/V, C++, C, and GeODE, a visual development environment. All applications, regardless of language, can have simultaneous access to the same database objects. In addition, GemStone uses a dialect of Smalltalk as an internal DML, which allows execution of methods or entire applications at the GemStone server. Supported platforms for GemStone, GeODE, and all language interfaces include UNIX workstations and servers from Sun, HP, IBM, and Sequent. Client-only support is available in a number of languages for Windows 3.1, OS/2 and Macintosh. Servio is an active member in the Object Management Group (OMG), the Object Database Management Group (ODMG), and the ANSI Smalltalk standardization committee.

3.1.1 Object model

The GemStone object model is essentially that of Smalltalk [GOLD83] (although GemStone itself is written in C and C++). Objects in GemStone consist of collections of instance variables and methods. Each object stored in GemStone contains an object identifier (called an *oop* for *object-oriented pointer*). As in Smalltalk, explicit object deletion is not supported. Instead, objects are automatically garbage-collected when they are no longer referenced by other objects. The GemStone ODBMS directly implements a variant of Smalltalk, Smalltalk DB (formerly called OPAL), which is specialized for database processing. For example, the language adds facilities for persistent objects, concurrent access, transactions, and indexing. As in conventional Smalltalk, Smalltalk DB provides both class definition (DDL) facilities and method definition (DML) facilities, as well as an extensive library of kernel classes. Because GemStone implements its own computationally-complete database programming language, GemStone stores complete objects (including both state and behavior) in the database, and allows the methods of these objects to be executed directly by the ODBMS.

An example object class definition in GemStone is[1]:

```
Object  subclass: 'Person'
   instVarNames:  #('priv_age' 'priv_spouse' 'priv_children')
   classVars: #(  )
   poolDictionaries:  #[  ]
   inDictionary:  UserGlobals
   constraints:  #[ #[#priv_age, Integer]]
   isInvariant:  false.

method: Person                   // age getter
   age
         ^priv_age %

method: Person                   // age setter
   age: aAge
         priv_age := aAge %
```

Similarly, a Part class might be defined as:

```
Object  subclass: 'Part'
   instVarNames:  #('privpartID' 'privpartName' 'privsubParts'
               'privisPartOf' 'privgraphic')
   classVars: #(  )
   poolDictionaries:  #[  ]
   inDictionary:  UserGlobals
   constraints:  #[ #[#privpartID, Integer],
                      #[#privpartName, String]]
       isInvariant:  false.
```

[1] Examples in this report are intended only to illustrate the general characteristics of syntax in the various systems described, and have not actually been tested on the target systems. As a result, the syntax is not guaranteed to be totally accurate or complete.

The `instVarNames` part of the declaration specifies the literal names of the instance variables of the object. In this case, there are five:

- `privpartID`, containing the oop of an integer object
- `privpartName`, containing the oop of a character string
- `privsubParts`, containing the oop of a set object containing the subparts of the part
- `privisPartOf`, containing the oop of a set object containing the parts this part is a part of
- `privgraphic`, containing the oop of an object of type `GraphicalObject` which is the iconic representation of the part.

(Each of these names starts with `priv` to indicate that it is private to the object, since instance variables are not visible outside the object.)

It would also be necessary to define the object class of objects containing sets of parts (these objects would serve as values of the `privsubParts` and `privisPartOf` instance variables defined above):

```
Set  subclass: 'PartSet'
       instVarNames:  #( )
       classVars: #(  )
       poolDictionaries:  #[  ]
       inDictionary:  UserGlobals
       constraints:  Part
       isInvariant:  false.
```

Instance variables are not necessarily constrained to contain only objects of specific types. The `constraints` part of the declaration can be used to specify such constraints. In class `Part`, only two instance variables are constrained: `privpartID` is constrained to contain only `Integer` objects, and `privpartName` is constrained to contain only `String` objects. In class `PartSet`, the set is constrained to contain only `Part` objects. This constraint mechanism is not intended as a general-purpose type-checking facility, but rather is intended to constrain instance variables over which indexes will be constructed to allow rapid retrieval. (The remaining parts of the declarations, such as `classVars`, etc., are irrelevant to this discussion, but were included for completeness.)

Method declarations are used to specify the externally accessible behavior of the object. For instance variables that represent externally visible object attributes, a pair of methods is defined for each instance variable, one to access it, and one to assign a value to it. For example, in class `Part`, the accessing and assignment methods for the `privpartName` and `privsubParts` instance variables are:

```
method: Part
   partName
         ^privpartName %

method: Part
   partName: aPartName
         privpartName := aPartName %

method: Part
   subParts
         ^privsubParts %
```

```
method: Part
   subParts: aPartSet
         privsubParts := aPartSet%
```

(In these methods, **^** denotes "return the value of", **:=** denotes ordinary assignment, and % is a terminator. Note that many methods are very small).

The classes also inherit certain methods from their superclasses. For example, both `Part` and `PartSet` classes inherit the `new` method for creating new instances. `PartSet` inherits methods for adding and removing `Part` objects from its superclass `Set`.

Once the classes are defined, objects can be created and manipulated using the methods. For example, if an object `myPart` exists in the database, the message

```
myPart  partName
```

asks `myPart` to return its part name. `myPart` is referred to as the message's *receiver* (the object to which the message is addressed), while `partName` is the message's *selector* (specifying the method to be executed in response to the message; see Section 2 for further discussion of this terminology).

Class `PartSet` inherits special methods from its superclasses (specifically, class `Collection`, a superclass of class `Set`) allowing it to be searched for objects with instance variables satisfying certain conditions. For example, the following expression would return all of the `Part` objects that are AND gates from a `PartSet` object `myParts`:

```
myParts select: [:aPart | aPart partName = 'AND gate']
```

GemStone also provides facilities for tuning the performance of object access. For example, class `Object` defines the basic clustering message `cluster` which is inherited by all objects. The simplest clustering method simply assigns the receiving object to a disk page--it does not attempt to cluster related objects pointed to by the receiving object's instance variables. Special clustering methods can be defined for individual object classes. For example, the following defines a method `clusterPart` for objects of class `Part`:

```
method: Part
   clusterPart
         self cluster.
         privpartID cluster.
         privpartName cluster.
         privsubParts cluster.
         privisPartOf cluster.
         privgraphic cluster.
         ^false%
```

The message `myPart  clusterPart` would then cause the `Part` object `myPart` to cluster itself. In this case, the method first asks `myPart` to assign itself to a disk page (`self cluster` does this), and then asks other objects pointed to by specified instance variables to cluster themselves on the same page (or a nearby page) as `myPart` (for example, `partName  cluster` asks the `String` object pointed to by `myPart`'s `partName` instance variable to cluster itself near `myPart`).

3.1.2 Architectural Features

GemStone uses a client/server architecture consisting of the following key components [BOS91]:

- *Gem* server (object manager) processes. Each Gem process is capable of executing object behavior specified in Smalltalk DB, the GemStone DML, and handles all query evaluation. The Gem server also contains both object and page caches. Each client process is associated with a Gem server, either by direct linking, or remotely.

- A *Stone* monitor process. Stone allocates new object identifiers, handles user login/out operations, coordinates transaction commit activity by multiple Gems, and handles recovery. The Gem processes communicate with Stone through interprocess communication mechanisms.

- *Page server* processes. Page servers provide Gem servers with page-level access to local and remote files. The page servers support databases distributed over multiple extents on different network nodes. A remote extent may also be locally replicated to improve performance.

- *Applications* (clients). Applications are developed using either the GemStone Smalltalk, C, or C++ interfaces, or the GeODE development environment, described below. An application is associated with a Gem server, and may be either linked directly to it, or run as a separate process.

These components may be organized in different configurations based on application requirements (Figure 3.1 shows one example, adapted from one in [BOS91]). For example, Gem (object management), page server, and Stone could be hosted on the same machine, resulting in a "server-heavy" configuration, Gem could be hosted on the same machine with the application, resulting in a "client-heavy" configuration, or could be on a separate machine of its own. A GemStone configuration may include multiple hosts in the network, and clients can simultaneously access databases on multiple hosts. GemStone also supports data replication, allowing critical objects to be redundantly stored on different servers.

GemStone can take advantage of the parallel processing capabilities of Symmetrical Multi-Processor (SMP) machines to improve throughput. GemStone can also take advantage of clustered CPU environments in which several servers are attached to a single disk subsystem.

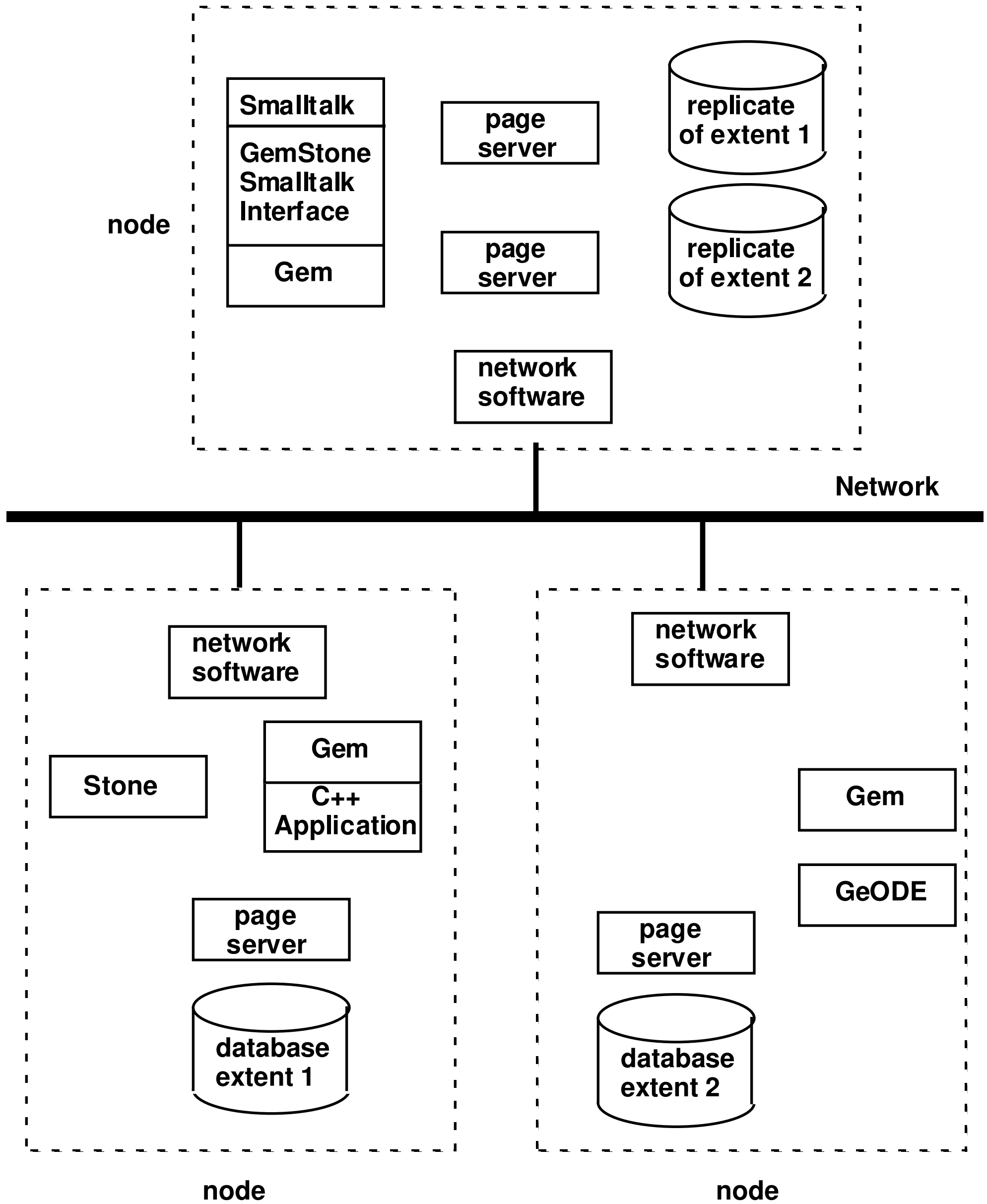


Figure 3.1 GemStone Client-Server Configuration

Because application processing can be programmed directly in Smalltalk DB, developers can choose how to distribute computation between Smalltalk DB computation on the host computer and Smalltalk, C++. or C computation on the application workstation. This *active ODBMS* (see Section 6.3.1) ability to execute methods in the database can be used to reduce network traffic and allow applications to take advantage of what may be the superior compute power of a database server. This can also eliminate the need to rebuild and re-deploy applications whenever application processing requirements or business rules change.

The GemStone C, C++, and Smalltalk Interfaces provide facilities necessary to allow external application programs written in C, C++, or Smalltalk to access GemStone (and its contained objects) while maintaining internal object consistency within GemStone.

The GemStone Smalltalk interface provides a set of Smalltalk classes that allow a Smalltalk application to communicate in a relatively seamless fashion with a GemStone database. The interface provides automatic mapping between most Smalltalk kernel classes and corresponding GemStone classes, and additional class mappings can be defined. The interface can automatically generate GemStone classes from Smalltalk application classes, and can similarly automatically generate Smalltalk classes from GemStone classes. The interface also maintains relationships between GemStone database objects and application objects. If either a Smalltalk application object or its GemStone counterpart is modified, the modification is automatically and transparently reflected in both locations.

Two versions of the GemStone Smalltalk interface are available. One is a linked version, in which the database session is linked into the same process space as the Smalltalk client for maximum performance. The other is an RPC version in which the interface communicates with remote database sessions, and is intended for distributed configurations.

The GemStone C interface is a library of C functions that provide a bridge between an application's C code and the objects in a GemStone database. A C program can access the data either structurally, by importing object data into the program, or by sending messages to invoke object methods in the GemStone database. Because C is not object-oriented, objects imported from the GemStone database into the C program must be represented as C data types, such as pointers, strings, or integers. Both an object's oop and a representation of the object's state may be imported into a C application's address space. However, access to objects directly in C circumvents object encapsulation, and must be done with caution. If the C program alters the structure of an object, the object may no longer behave as expected. A C program can use only predefined oops, or oops it has received from GemStone. The program cannot create new oops directly, since these might conflict with oops already existing in GemStone. Functions are available to import and export individual objects of the various classes.

The C++ interface is based on the use of a C++ preprocessor, together with:

- the <u>GemStone C++ Class Library</u>, an application-oriented subset of the full GemStone Class Library (which in turn is based on the Smalltalk Class Library). It can be used directly in C++ programs to create persistent objects, or to derive additional persistent classes.

- the <u>Registrar</u>, a utility that creates the runtime interface between the C++ program and the GemStone database. Specifically, it "registers" persistent C++ classes in the GemStone database, and generates the C++ code and information necessary to transfer objects to and from GemStone.

In using the interface, persistent C++ classes are declared in a header file using standard C++ syntax. Classes derived directly or indirectly from a persistent GemStone class (either a class in the GemStone C++ Class Library or a previously-registered persistent class) are automatically made persistent. Class data members can include C++ kernel types, user-defined classes, classes from the GemStone C++ Class Library (such as `GS_Object` and `GS_Set`), and one-dimensional arrays of these classes. The header file is input to the Registrar utility, which "registers" any persistent classes by creating a corresponding class in the GemStone database. The Registrar also produces output files containing translation procedures and mapping information used to move objects between the C++ application and the GemStone database. These files are compiled and linked with any C++ programs that will use the persistent classes. The interface also includes functions to establish and terminate database sessions, move objects between the C++ program and the GemStone database, commit and abort transactions, etc.

Instances of C++ classes registered with GemStone can be referenced using two different types of pointers. GPointers are object identifiers, and are transparently dereferenced. DPointers (direct pointers) are ordinary C++ pointers (virtual memory addresses). Further details concerning the differences between these types of pointers, and other aspects of a C++ ODBMS interface, are provided in Section 3.2's discussion of the ONTOS DB C++ interface, and in Section 6.1.

A C++ application can concurrently access and modify GemStone objects created by other languages, including C, Smalltalk, or any language that makes C calls. Similarly, a Smalltalk application can concurrently access and modify GemStone objects created by C++, C, or any language that makes C calls. New server methods can be added to GemStone at any time. These methods can be written in either Smalltalk DB or C, and may be called either by other server-based methods or directly by applications.

GemStone provides both *optimistic* and *pessimistic* concurrency control mechanisms. The optimistic mechanism works roughly as follows. A Smalltalk DB transaction initially operates on a private copy of GemStone's objects called a *workspace*. GemStone tries to merge any modified objects in the workspace with the main, shared database only when the transaction tries to commit. At that time, GemStone checks to see whether other users have, since the transaction began, committed transactions of their own that may have changed the objects read or modified by the first transaction. If they have, GemStone refuses to commit the transaction, since the transaction may have used some outdated values in computing its modifications. When GemStone refuses to commit a transaction, it leaves the workspace intact. The application can then perform any desired cleanup or saving of results created during the transaction and abort the transaction. This begins a new transaction and creates a new workspace containing up-to-date copies of shared, committed objects. The transaction can then be reattempted using the up-to-date workspace. A read-only transaction can always commit, since it has created no workspace modifications. Such an optimistic mechanism generally provides more concurrency when there are likely to be few actual conflicts among users. GemStone also provides a conventional (or *pessimistic*) concurrency control mechanism based on locking. Using this mechanism potentially reduces concurrency, but insures that commits of database modifications will succeed. An application can submit transactions that use either approach: objects that are unlikely to produce conflict can be accessed optimistically, thus avoiding the overhead (and possibly loss of concurrency) due to locking, while transactions that are likely to encounter conflict can use the pessimistic mechanism.

GemStone provides optimized queue and collection classes that minimize concurrency conflicts in applications where high concurrency is required (Section 2.2.6 discussed how such a queue could be defined). Reduced conflict indexes, bags, counters, and hashed-access dictionaries are also provided. GemStone also supports object change notification, in which the database can notify an application when a specified object has changed, and allows applications to send signals to other applications without writing data to the database. This signaling mechanism can be used to implement strategies that involve *cooperation* among multiple applications.

GemStone supports a versioning mechanism. This mechanism is used to support several predefined versioning policy options (e.g., linear versioning), and can also be used to define application-specific versioning policies (see Section 6.5 for a discussion of versioning). GemStone supports schema modification through class versioning and allows full migration of objects between versions of their classes. Migration is fully customizable and is undoable.

Servio also offers GeODE (GemStone Object Development Environment). GeODE is a comprehensive environment for rapidly designing, building and deploying object applications. Its design promotes code reuse in a team programming environment for increased productivity. With GeODE's visual programming tools, programming an application is done by wiring together graphical representations of encapsulated code blocks. A simple extension mechanism promotes the re-use of code, increasing the speed of program development. GeODE applications are stored and run in the GemStone database, and so are both self-porting and network-aware, and can be accessed externally from any of the GemStone language interfaces. Because of GemStone's network architecture, GeODE applications can operate easily in a client/server environment.

GeODE consists of six main elements:

- Visual Application Manager: Provides centralized management of each application and its component parts, and a namespace for addressing known objects. The Application Manager can be used to store Visual Program Designer-developed applications in the GemStone ODBMS as executable objects.

- Visual Schema Designer: Allows the development of database schema visually, making the process more interactive and intuitive than with object-oriented programming languages. It also provides analysis tools for examining an existing schema.

- Visual Forms Designer: The Forms Designer reads GemStone class definitions and an associated data dictionary to automatically create default forms suitable for simple data entry. These forms can be rapidly customized, using a wide selection of user interface components and field types, which include image and sound support, and a large set of form design aids. The list of field types can be extended interactively.

- Visual Program Designer: The Visual Program Designer allows developers to visually create and modify the behavior of an application without having to write code. Programs are created graphically on a screen by connecting visual program blocks (code chunks) to field blocks drawn from the forms created in the Forms Designer. Predefined program blocks include constructs such as iterators, conditionals, adding values, defining GUI objects such as buttons, etc., and users can extend the catalog in any of a number of simple ways. Smalltalk DB code can be integrated easily.

- Developer Class Library: GeODE comes with more than 480 classes and thousands of methods, and is easily extended for handling specialized applications. In a team environment, some programmers can develop visual applications while others write new methods that are encapsulated into visual program blocks for easy reuse.

- Developer Tools: GeODE includes tools for debugging, browsing and inspecting applications. Included in this set of tools are several debuggers, browsers, inspectors, an object clipboard, an image editor, and a code profiler for performance analysis.

The GemStone DataBridge Toolkit allows existing data stored in relational databases to be integrated with GemStone data. GemStone applications and Smalltalk DB methods can send SQL queries to a relational DBMS on the same network. The results of a query are returned as a set of GemStone objects for further processing within GemStone.

3.2 ONTOS DB

ONTOS DB [ONT94a,b] is Ontos, Inc.'s second-generation ODBMS product (the first generation product was called Vbase [AH87]). ONTOS DB is based on C++, and tries to combine C++ programming capabilities with the persistence, sharing, integrity, and query capabilities of a DBMS. ONTOS DB uses a client-server architecture, and the content of an ONTOS DB database may be physically distributed over any number of nodes in a network. ONTOS DB comes with a C++ interface, allowing the use of standard C++ compilers to access ONTOS DB databases, an SQL interface, allowing SQL access to ONTOS DB both interactively and via programs, and an interactive graphical interface. ONTOS DB runs on Unix workstations, including those from Sun, HP, and IBM, as well as on OS/2. ONTOS DB's concurrency control and transaction mechanisms support consistent database access by multiple concurrent, and potentially-conflicting, users. In addition, special features are included to promote data sharing among *cooperating* processes.

3.2.1 Object model

ONTOS DB uses an internal object model which is mapped to the C++ object model for application access. Object properties and functions are defined in C++ (with minor extensions). ONTOS DB objects consist of collections of *properties* and *functions*. Each object is identified by a 64-bit object identifier or *OID*, which is guaranteed to be unique and consistent within an ONTOS DB database. Each object's properties are either stored directly in the object or represented by OID references to other objects, allowing the direct representation of complex object relationships. ONTOS DB also provides aggregate classes for modeling one-to-many relationships. It resolves references directly and presents objects that are fully C++-compatible at the C++ interface.

ONTOS DB provides database functionality through the ONTOS DB class library. Some of the classes in this library are shown in Figure 3.2.1. A key object class in the library is class `OC_Object`. This is the parent of all persistent object classes (all user-defined classes that are to be persistent must inherit from it). Class `OC_Object` defines a constructor function for creating objects in the database, and a destructor function for deleting them (unlike GemStone, which follows Smalltalk in using garbage collection to automatically delete unreferenced objects, ONTOS DB follows C++ in requiring explicit deletion of objects that are no longer needed). Class `OC_Object` also defines properties for an object name, and for the physical clustering of objects in the database. Like GemStone, ONTOS DB also provides a number of generally useful aggregate classes, such as OC_`Dictionary`, OC_`Set`, OC_`Array`, and OC_`List`. For example, dictionaries are implemented as hash tables or B* trees. These data structures are available as general abstractions, eliminating the programming ordinarily required to create, access, and maintain them.

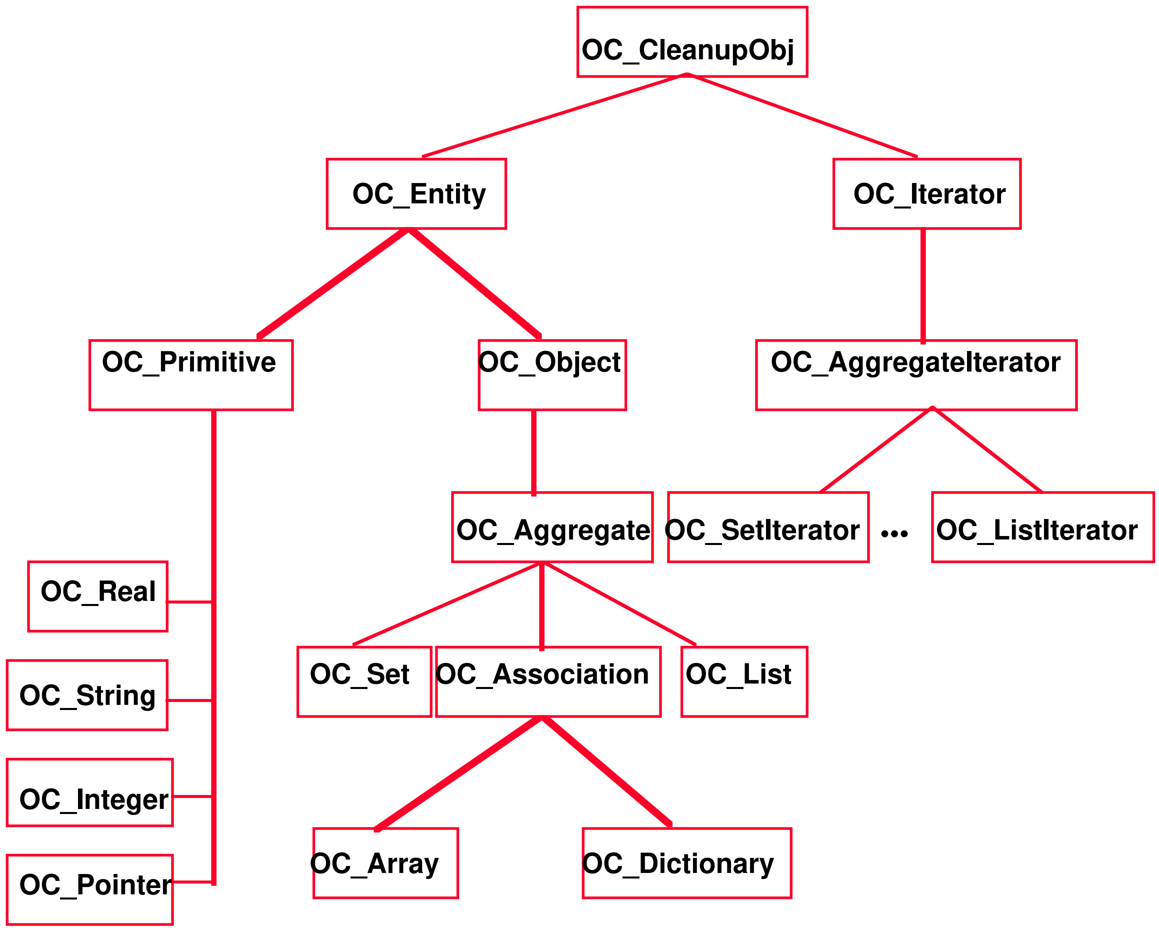


Figure 3.2.1 Part of ONTOS Class Hierarchy

Class `OC_CleanupObj` is part of the exception handler mechanism; it defines the properties objects need to be cleaned up following an abort. Instances of class `OC_Primitive` are like values. They are treated as objects, but do not have the full functionality of objects (e.g., member functions cannot be defined for them, and they cannot be inherited from).

An example C++ object class definition in ONTOS DB is:

```
class Person : public OC_Object {
{
private:
   int              priv_age;
   OC_Reference     priv_spouse;
   OC_Set*          priv_children;

public:             // constructors and get direct type
   Person(char* name=(char*)0, int priv_age=0);
   Person(OC_APL* theAPL);
   virtual OC_Type* getDirectType();

                    // accessor functions
```

```
    int        age() { return priv_age; }
    void       age(int age) { priv_age = age; }

    Person*    spouse( )
         { return (Person*) priv_spouse.getReferent(this); }
         // getReferent activates the spouse object
    void       spouse(Person *aPerson)
         { priv_spouse.reset(aPerson,this); }
         // reset makes a Reference to the new spouse object

    void       display();

  // functions for dealing with children //
    ...
  };
```

In the above declarations, class `Person` is defined as inheriting from class `OC_Object`, which makes it an ONTOS DB persistent class. The private declarations define the internal state of the object. These three variables are not visible outside the object. The variable `priv_children` is defined as type OC_`Set*`, meaning first that the value is a pointer to an `OC_Set` object, and second that the pointer is an ONTOS DB *direct* pointer. On the other hand, the variable `priv_spouse` is defined as type OC_`Reference`, which means that the value is an ONTOS DB *abstract* pointer (the distinction between these types of pointers is discussed below). The object also inherits certain instance variables from its superclasses. For example, it inherits the instance variable `name` from class `Object`. This is used to allow initial access to the object when its object identifier is not known.

The public part of the declaration specifies the externally accessible C++ member functions (methods) for the object. In defining an ONTOS DB persistent class, two object constructor functions (named `Person`) must be specified. The first public function specified above is an ordinary C++ object constructor. The second public function is an ONTOS DB activation constructor, which is used for reconstructing objects read from disk. The third function redefines the standard member function `getDirectType()`, which returns a database object representing the type of the object. In addition, there is a function to access each instance variable (the ones for dealing with children are omitted). The object also inherits certain functions from its superclasses. For example, it inherits functions `deleteObject`, `putObject`, and `Print` (among others) from class `Object`.

In C++, as in many programming languages, there is no standard mechanism for representing class data definitions for runtime use. The ONTOS DB class library provides classes for expressing properties, member functions and their arguments, and the overall class definitions themselves. These classes are used to represent schema definitions to the database in a machine-independent way, and are also useful in program development when runtime knowledge of the structure of an object is needed. Instances of these classes representing user-defined objects are generated automatically from C++ declarations by a utility called *classify*.

Among the schema representation classes are classes needed to express member function declarations. These classes allow key algorithms to be packaged in the database rather than with the application code. The behavior of the application can then be changed without changing the application programs themselves. For example, a text processing application might support multiple natural languages by placing all language-dependent functions into the database.

The general concepts of object addressing and clustering in ONTOS DB are similar to those of GemStone in many respects. Like GemStone, ONTOS DB provides random access within a one-dimensional address space based on object identifiers (OIDs). The granularity of reference is a single object. Aggregates (container objects) are themselves denotable as objects. Property values (except in some special cases) are stored in the form of references to other objects. This is ideal for applications such as those discussed in Section 2, in which the data structures used are complex graph structures whose edges are frequently implemented in applications by direct memory references. ONTOS DB is also convenient for such implementations because it supports direct translation between virtual memory references and object OIDs.

Like GemStone, ONTOS DB also supports the clustering of related objects. For example, in a CAD application it may be common to access a part object and the object representing its image property at the same time. Clustering requirements are usually application-dependent, and must be determined by observing or predicting the access patterns of specific applications. ONTOS DB allows the programmer to specify clustering and provides tools for reclustering when more experience with the application permits better choices to be made.

The general steps in setting up an ONTOS DB application are as follows. First, the *DBATool* utility would be used to register the database with ONTOS. Second, the application code would be designed and implemented. This would include the specification of the C++ object classes required by the application. Third, the *classify* utility would be used to translate the C++ headers for the class definitions into persistent database class definitions (this corresponds to the GemStone C++ interface *Registrar* utility), as illustrated in Figure 3.2.2. Finally, the code would be compiled and linked using the *cplus* C++ preprocessor utility (see below) and the ONTOS DB library.

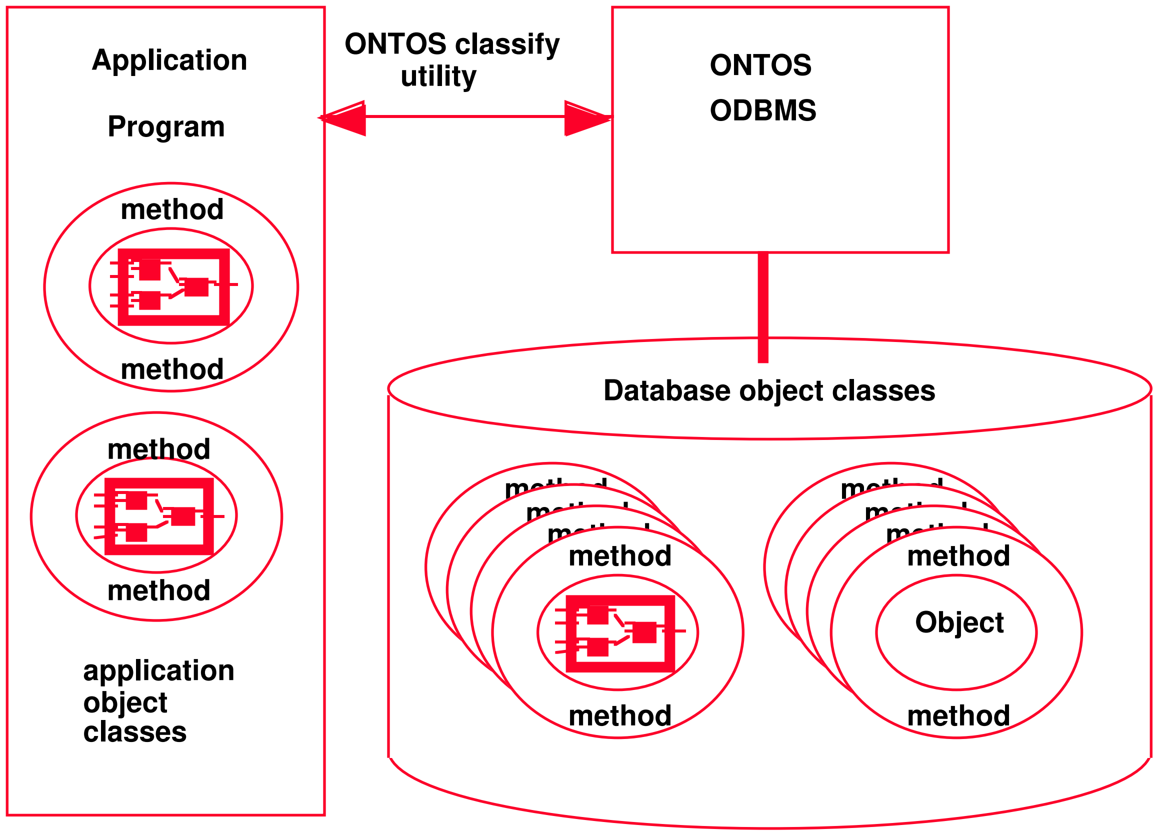


Figure 3.2.2 Registering Database Classes

3.2.2 Architectural Features

The ONTOS DB distributed client-server architecture is shown in Figure 3.2.3 [ONT94a]. In this architecture, the server side manages the data store, while the client side provides the interface to user processes and manages the mapping of data to the application program's virtual memory space. In a LAN installation, the client and server sides communicate over the LAN. When both reside on the same node, direct interprocess communication is used instead. A third component of the architecture is the Registry. The Registry keeps track of all databases in the network, and their primary servers. It also maintains user profiles and access rights for database security. The Binder is a network-wide service responsible for establishing connections between clients opening a logical database and the servers that manage the areas of that database. The Binder consults the Registry to identify and locate servers associated with the requested logical database. A logical database consists of one or more *areas*.

A database may be contained on a single node or distributed over a number of nodes in the network. Each database is controlled by a primary server. If the database is distributed over other nodes, each of these nodes has a secondary server managing local storage. The task of each server process is to manage its local part of the database, and to respond to client requests over the network. In addition, the primary server maps objects to their respective servers. It is also responsible for all operations global to the database. These include open and close requests, and the control of multi-server transactions.

The servers of a database are responsible for managing it as a logical object store. Each object in a database is identified by an identifier (OID) guaranteed to be unique within the database. Each database, whether contained on a single node or distributed over several nodes, has a single OID space. Objects may also have names. Names are mapped to OIDs by a hierarchy of name directories.

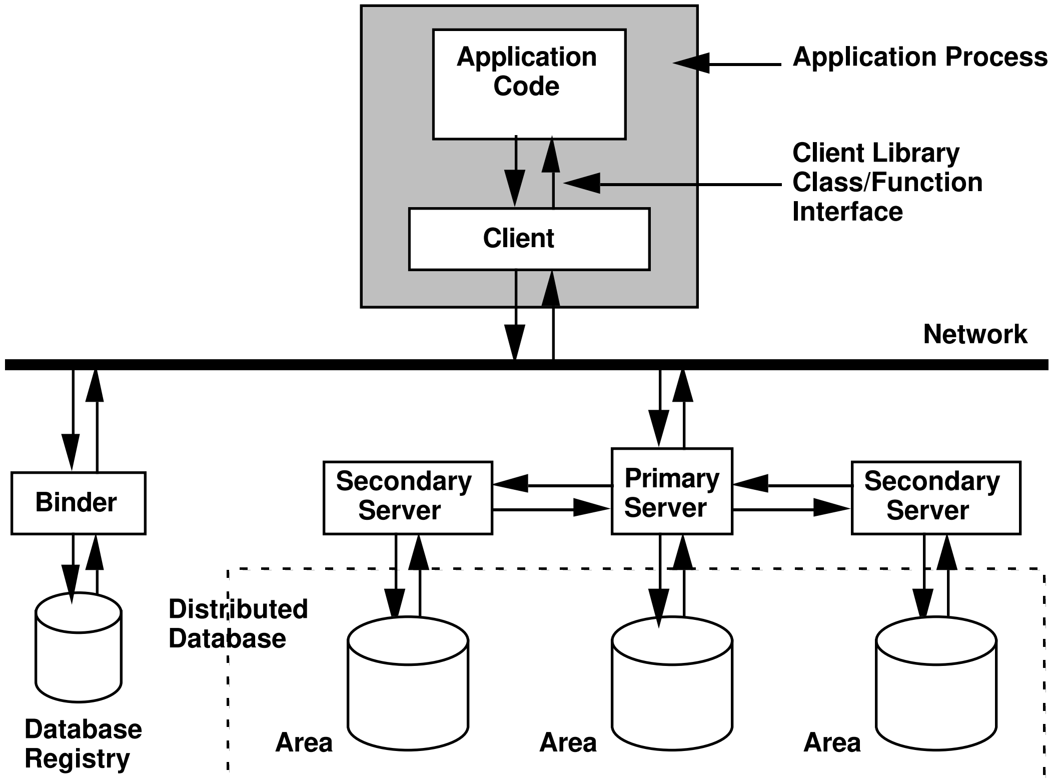


Figure 3.2.3 ONTOS DB Client-Server Architecture

A C++ application process contains the application code and the client portion of ONTOS DB. The client portion is implemented as a function and class library (the ONTOS DB Client Library) that is linked into the application when it is created. The client manages communication between the application and one or more servers over the network. It also provides the application with various storage management services, and translates object references between their OID (database) form and virtual memory addresses used during processing. The client is designed so that the application can take over some of these services (such as memory allocation) in cases where the application is able to make application-specific optimizations. The interface between the application and the client is composed of a relatively small set of functions and classes. The major functions are database `Open` and `Close`, transaction `Start`, `Commit`, and `Abort` operations, and the object I/O functions.

Objects requested singularly or in groups by the client are retrieved by the servers and handed back to the client. These objects are *activated* into the client application's virtual memory and manipulated as C++ structures. When the application finishes with these objects, it *deactivates* them, optionally deallocates their memory, and passes them to the client. The client portion transfers them back to the servers, which apply the changes made to objects to the areas.

Low-level storage facilities are provided for ONTOS DB objects by *storage managers*. These components handle an object's disk and memory storage by allocating and deallocating space and storing and deleting objects. A storage manager also resolves and specifies `OC_References` to objects and performs reference resolution operations. Storage managers are provided as instances of class `OC_StorageManager` or its subclasses. ONTOS DB provides a set of storage managers that are optimized for different types of data and storage requirements. These are provided as `OC_StorageManager` subclasses, so that they can be customized by users to meet installation-specific requirements. Specifically:

- The standard storage manager (class `OC_StandardSM`) handles activation, clustering, and locking of individual objects (e.g., paragraphs in a document management application). This is the default storage manager class.

- The group storage manager (class `OC_Group`) handles activation, clustering, and locking of fixed-size units called *group pages*. A group page is used to hold instances of objects of a single type. The main benefit of this grouping is its activation and deactivation performance, which is optimal for large numbers of small objects that are typically accessed and locked in groups (e.g., sets of coordinates in a CAD or GIS application). Activation of an object stored in a group page results in the activation of all objects stored in that page.

- The in-memory storage manager (class `OC_InMemorySM`) is used internally by ONTOS DB to support creation of temporary, non-persistent objects, which can optionally be converted into database objects. Users cannot create instances of this class or call it directly, but they can retrieve pointers to instances of it created by ONTOS DB.

- The version storage manager (class `OC_VersionSM`) controls access to versions of objects by providing reference resolution and configuration management support. This storage manager delegates the physical and in-memory storage management aspects of its functions to other storage managers, such as the standard storage manager.

- The external storage manager (class `OC_ExternalSM`) and related components are provided as an optional product in Release 3.0. The external storage manager components provide facilities for managing external data stores, such as relational or hierarchical DBMSs and proprietary file formats, and mapping and converting data in those external stores into objects. Because these components are provided as object classes, they can be tailored to the specific requirements of different installations and external stores.

Ontos intends to provide specific specialized instances of external storage managers as off-the-shelf products, with Sybase and IMS versions among those planned.

The skeleton of a simple ONTOS DB application is shown below. It illustrates a number of key points about the use of ONTOS DB (and the C++ interfaces to a number of other ODBMSs).

```
OC_open("personDB");                          //open the database

OC_transactionStart();                        //start a transaction
```

```
Person *myPerson = (Person*)OC_lookup("Frank"); //activate a Person object

<code to modify the Person object>              //modify myPerson

myPerson->putObject();                          //deactivate myPerson

OC_transactionCommit();                         //commit the transaction

OC_close();                                     close the database
```

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. In this case, an initial object is activated using the `OC_lookup` function, which activates an object given its name. Named objects constitute entry points into the database from applications. Database objects use OIDs for inter-object references. All references contained by the activated object to other already-read objects are translated 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. Because these processes can critically affect performance, the number of objects activated is placed under the control of the programmer, who can choose to activate a single object, or a logical cluster of objects. The clustering of objects is under application control to allow objects frequently used together to be stored and retrieved in a single database operation. All database operations are performed from within transactions, in order to guarantee consistent access when the database is being shared by applications that may be performing conflicting operations on 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, 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), it is activated automatically by the system. Activation is thus transparent to the programmer, at the cost of additional indirection. Abstract references can be used to guarantee reference safety when individual objects or clusters are activated, and is expected to be the most common form of referencing. 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 `OC_lookup`, as noted above. Facilities corresponding to ONTOS DB's direct and abstract references are available in a number of the C++ ODBMS interfaces described in this report (see Section 6.1 for a further discussion of this topic).

The ONTOS DB transaction mechanism contains a number of enhancements over conventional database transaction mechanisms. First, it allows for *nested transactions*. Each level of nesting is atomic on its own. This makes it much easier to provide a general *undo* facility. ONTOS DB also supports *cooperating processes*. Multiple processes may join in a single transaction. Such a shared transaction combines the changes made by multiple processes into a single atomic change while providing the processes with a common data pool for sharing and exchanging information. ONTOS DB, like GemStone, also supports both optimistic and pessimistic concurrency control, at the programmer's option. In addition, ONTOS DB's High Concurrency Object (HCO) Technology provides an extensible database concurrency mechanism. HCOs enable user-defined concurrency schemes that support simultaneous same-object updates (multiple writers).

The ONTOS DB Object SQL interface is intended to add a predicate-based style of interaction with the database to the navigational style characteristic of C++. OSQL is intended for use by application programmers, and is intended both to allow application programs to issue database queries, and as a back-end for an interactive end-user query facility. ONTOS DB Object SQL extends SQL to allow the unique capabilities of an object model to be accessed via SQL. For example, object systems frequently model 1-n relationships with aggregate objects rather than through join operations. SQL must be extended to handle such relationships. In addition to accepting class names (the object analog of tables in relational systems), the FROM clause in Object SQL will accept any argument that evaluates to a collection of objects. For instance, it will search aggregates of the class, iterators defined for such aggregates, or an explicit list of objects. Similarly, the SELECT clause will accept property names (analogous to relational columns) as well as member function invocations and "navigational-style" property-chain expressions. An example of the latter is `person.mother.mother.name` for grandmother's name. Finally, the WHERE clause is extended to allow arbitrary Boolean expressions, including member function calls that return Boolean values.

Class *extensions* must be declared for ONTOS DB types and instances to be used in OSQL queries. An extension is the collection of all of the instances of a type that exist at a given time. Extensions are set automatically when types are created using the DBDesigner tool. Types can be explicitly specified as having extensions when defined programmatically or when using the *classify* utility.

A *query iterator* object is used to implement an OSQL query. Like other iterators in ONTOS DB, the query iterator uses a special operator to successively return each of the objects that meet the query conditions. The following is the outline of a simple OSQL query session:

```
OC_startQuerySession();
...
OC_QueryIterator* aQuery;
...
aQuery = new OC_QueryIterator("select address.state, address.zip,
manager from employees");

while (aQuery->moreData())
    {
    aQuery.yieldRow (state, zipcode, mgr);

    <process data>

    }
...
OC_endQuerySession();
```

In this case, an Object SQL string is specified which is used to create the `OC_QueryIterator` object `aQuery`. The application then iterates over this object using `yieldRow()` to return each successive row until the query iterator is empty. The extensions to SQL include path expressions that specify traversal of object references rather than relational joins, using a dot notation, as in:

```
select Name
from Employee
where Employee.Department.Location = "Bermuda"
```

The extensions also allow function expressions to be substituted for values anywhere a value is allowed, as in:

```
select Name
from GetSubParts(OC_lookup ("Engine Block"))
where Supplier = OC_lookup ("Henry Ford's Motor Parts")
```

(As noted above, `OC_lookup` is a predefined function for objects which looks up a object by name).

Specifically:

1. The SELECT clause (target list) in a query can reference

   - properties (data members)
   - object operations or separately-defined procedures
   - a navigational reference (path expression) to a property or procedure (path expressions can only use multi-valued properties at the ends of chains)
   - table column heads, arithmetic operations, wildcard operator (*)

2. The FROM clause in a query can reference:

   - a type name
   - an aggregate name
   - a navigational reference (path expression) to a type or aggregate
   - an explicit set of objects or types
   - a procedure invocation that returns a type or aggregate
   - an expression that includes any of the above

In addition, the WHERE clause in a query can include any expression that evaluates to a Boolean, as well as invocations of object operations and path expressions.

Object operations may have side-effects (it is the user's responsibility to be aware of them). Queries return a row of results for each satisfying object. The rows may be output as either:

- A character string (of C++ type `char*`).

- An object of type `OC_ArgumentList`. This list contains one object of type `OC_Argument` for each column. An `OC_Argument` object can be cast back to its original type.

- An iterator which, when invoked, sequentially returns the columns of the row as `OC_Argument` objects.

When the latter two forms are used, the query results may be referenced in subsequent queries.

Features of standard SQL *not* supported include:

- nested queries
- SQL security features
- SQL transaction management
- ROLLBACK and COMMIT
- data manipulation operations (INSERT, UPDATE, DELETE)
- cursor operations (this is provided by `OC_QueryIterator`)
- UNION
- existence tests
- GROUP BY, HAVING
- ORDER BY
- built-in functions: `sum`, `count`, `avg`, `min`, `max`

ONTOS DB also includes the following tools and utilities:

- *DBATool* provides an interface to the ONTOS DB Database Registry. It defines the mappings between one or more logical databases and the physical locations of the data across a network, as well as the agents that will manage access to the database. DBATool has a command line interface, an interactive interface, and an interactive graphical interface (called *Graphical DBATool*).

- The *classify* utility is the schema compiler used in ONTOS DB. It reads the class definitions in C++ header files and generates a corresponding database schema.

- The *cplus* utility is a preprocessor used to prepare C++ constructors, destructors, and member functions for use with ONTOS DB. *cplus* must be used to compile all modules linked to the ONTOS DB Client Library.

- *DBDesigner* is a graphical tool used for schema design and database browsing. It also provides automatic generation of C++ header files to implement designs created with the tool, and can also be used to browse over or modify existing objects, or create new objects.

- The schema evolution and instance migration (SEIM) utilities *miginit* and *migtool* are tools used to change the instances of a type to conform to changes made to the type's definition.

- Other utilities are also provided for journal analysis, and server performance analysis.

ONTOS recently announced plans to ship an NT port of its database by late this year, motivated by a large contract with a San Francisco-area utility company (*ComputerWorld*, 5/2/94). A three-way deal of Pacific Gas & Electric Co. (PG&E), ONTOS DB, and Microsoft calls for codevelopment of customer-service, billing, network management, and other applications for an unspecified number of years. The utility's network expansion calls for ONTOS DB 3.0, and specifically the use of the external storage manager facility to allow users to access non-object data sources.

3.3 ObjectStore

ObjectStore [LLOW91; ODI92a,b,93,94] is an ODBMS developed by Object Design Inc. (ODI). It implements a full object-oriented data model, and offers tightly-integrated language interfaces to a complete set of traditional DBMS features including persistence, transaction management (concurrency control and recovery), distributed access, associative queries over large amounts of data, and database administration utilities. The system supports programming language interfaces for C, C++, and Smalltalk, and comes with a graphical, interactive schema designer, incremental compilation and linking facilities, source-level debugging, and a graphical, interactive database browser. ObjectStore supports C++ extensions providing set-oriented data manipulation (query support). ObjectStore is supported on Sun, HP, IBM, DEC, NCR, Olivetti, and Silicon Graphics Unix platforms, VAX/VMS, and Windows 3.1, Windows NT, OS/2, and Novell Netware PC platforms. Object Design is an active participant in the Object Database Management Group, and in C++ standards activities.

3.3.1 Object Model

In general, the ObjectStore user sees the object model of the particular programming language interface being used. The ObjectStore C++ interface supports the object model of C++, including support for class hierarchies, object attributes and direct inter-object relationships, and user-defined encapsulated data representations. One of the goals of ObjectStore is to minimize the difficulty of altering native C and C++ applications to run over ObjectStore. A number of facilities in ObjectStore support this. For example, in ObjectStore, persistence is specified at the level of the individual object or variable, rather than at the type level, meaning that objects of any C++ data type can be allocated either transiently or persistently. It is not necessary to inherit from a special base class to obtain persistent behavior. This allows functions written to operate on non-persistent data to also operate on persistent data, and allows programs to be converted to run on ObjectStore relatively easily. It also allows the use of commercial C++ libraries with ObjectStore in a particularly straightforward way. This is because the class definitions in the programs or class libraries need not be altered to obtain persistent behavior for class instances.

ObjectStore also offers a number of extensions to C++. One such extension is the *collection* facility, which is provided an object class library. The collections supported by this library include sets, bags, lists, and arrays, and the classes include both parameterized (e.g., class `os_Set<employee*>` to represent a set of employees) and unparameterized (e.g., class `os_list`) forms. ObjectStore provides facilities to allow users to specify the intended usage of these collections (e.g., frequency of insertion), together with other tuning information, and will choose an appropriate representation based on this information. ObjectStore also provides a looping construct to iterate over these collections. This, together with ObjectStore's query facilities, will be discussed later in this section.

ObjectStore also provides support for bi-directional *relationships* between objects. A relationships is essentially a pair of inverse pointers between the related objects (each has a pointer pointing to the other). One-to-one, one-to-many, and many-to-many relationships can be defined. The relationship facility maintains the integrity of these relationships. For example, if one participant in a relationship is deleted, the inverse pointer is automatically set to null. More complex maintenance is also supported. For example, the skeleton declaration below illustrates an inverse relationship between employees and departments. In this example, if an employee e is added to department d, e's department would automatically be updated to d. Conversely, if e's department were updated to another department d2, e would automatically be removed from `d->employees` and inserted into `d2->employees`.

```
class employee
{
public:
...
   department* dept
      inverse_member department::employees;
};

class department
{
public:
...
   os_Set<employee*> employees
      inverse_member employee::dept;
};
```

Access to ObjectStore is provided through a library based application interface compatible with popular C and C++ compilers and programming environments. The classes in this interface (calls in the C interface) provide access to database functionality. ObjectStore also provides an extended C++ (ObjectStore DML) which provides a tighter language integration to the ObjectStore extensions, such as the query and relationship facilities. This interface is accessible only through ObjectStore's C++ compiler [Hal93].

[ODI94] describes ObjectStore's Smalltalk interface (currently in Beta test) for Release 3, which is compatible with ParcPlace VisualWorks Smalltalk. The interface provides database functionality through a combination of new Smalltalk classes and modifications of existing classes. Like the C++ interface, the Smalltalk interface is designed to provide database functionality in a way that is as transparent as possible to the Smalltalk programmer, and which naturally supports normal Smalltalk behavior. Databases are created by sending messages to class `ObjectDatabase`, as in

```
db := ObjectDatabase create: '/omg/parts'.
```

Class `ObjectDatabase` also supports other protocol for controlling the database as a whole.

Database access must always be within the scope of a transaction. The following example from [ODI94] illustrates a simple program which opens a database, accesses an existing department, creates a new persistent object of class `Employee` and initializes its instance variables, and adds the employee to the department.

```
| db dept emp |
db := ObjectDatabase open:'/company/records.db'.
ODBTransaction transact: [
   dept := db at: 'department'.
```

```
    (emp := Employee newIn: db)
        name: 'Fred'
        salary: 1000.
    dept addEmployee: emp
]
```

In ObjectStore's Smalltalk interface, persistence is orthogonal to type, meaning that:

- Only one set of classes must be written, and instances of any class can be either persistent or transient.

- No new instructions must be added for accessing persistent data.

In the example above, the statement `Employee newIn: db` specifies that a new persistent employee is to be created. The only difference from the usual Smalltalk `new` method is the `newIn` argument specifying the database in which the object is to be created.

ObjectStore's Smalltalk interface also provides for the *implicit* creation of persistent objects. Objects created during the scope of an ObjectStore transaction which are reachable only from persistent objects are automatically made persistent when the transaction commits. This approach allows existing user or library code to work properly for persistent objects with little or no change, because the changes required to add persistence can often be made outside the existing code. This can be done by creating one or more new persistent objects, and having them reference objects created by the existing or library code, in which case the implicit process makes the referenced objects persistent automatically. The approach also makes it possible for an application to allocate a few top-level objects explicitly, and rely on this implicit process to handle the persistence of any other objects automatically. Whether persistent or transient, objects are accessed using normal Smalltalk techniques.

3.3.2 Architectural Features

ObjectStore uses a client/server architecture that allows one server to support many client workstations, each workstation to simultaneously access multiple databases on many servers, and a server to be resident on the same machine as a client. This architecture is shown in Figure 3.3.

ObjectStore executes an ObjectStore server process on every machine which acts as a database server. The ObjectStore server provides the long-term repository for persistent data. Objects are stored on the server in the same format in which they are seen by the application in virtual memory, which avoids overhead in moving between different persistent and transient forms of the object. Objects can cross page boundaries, and can be much larger than a page. The server stores and retrieves pages of data in response to requests from clients. The server knows nothing about page contents; it simply passes pages to and from the client, and stores them on disk. The server is also responsible for concurrency control and recovery, using techniques similar to those used in conventional DBMSs. It provides two-phase locking with a read/write lock for each page. Recovery is based on a log, using a write-ahead log protocol. Transactions involving more than one server are coordinated using a two-phase commit protocol. The server also provides backup to long-term storage media such as tapes.

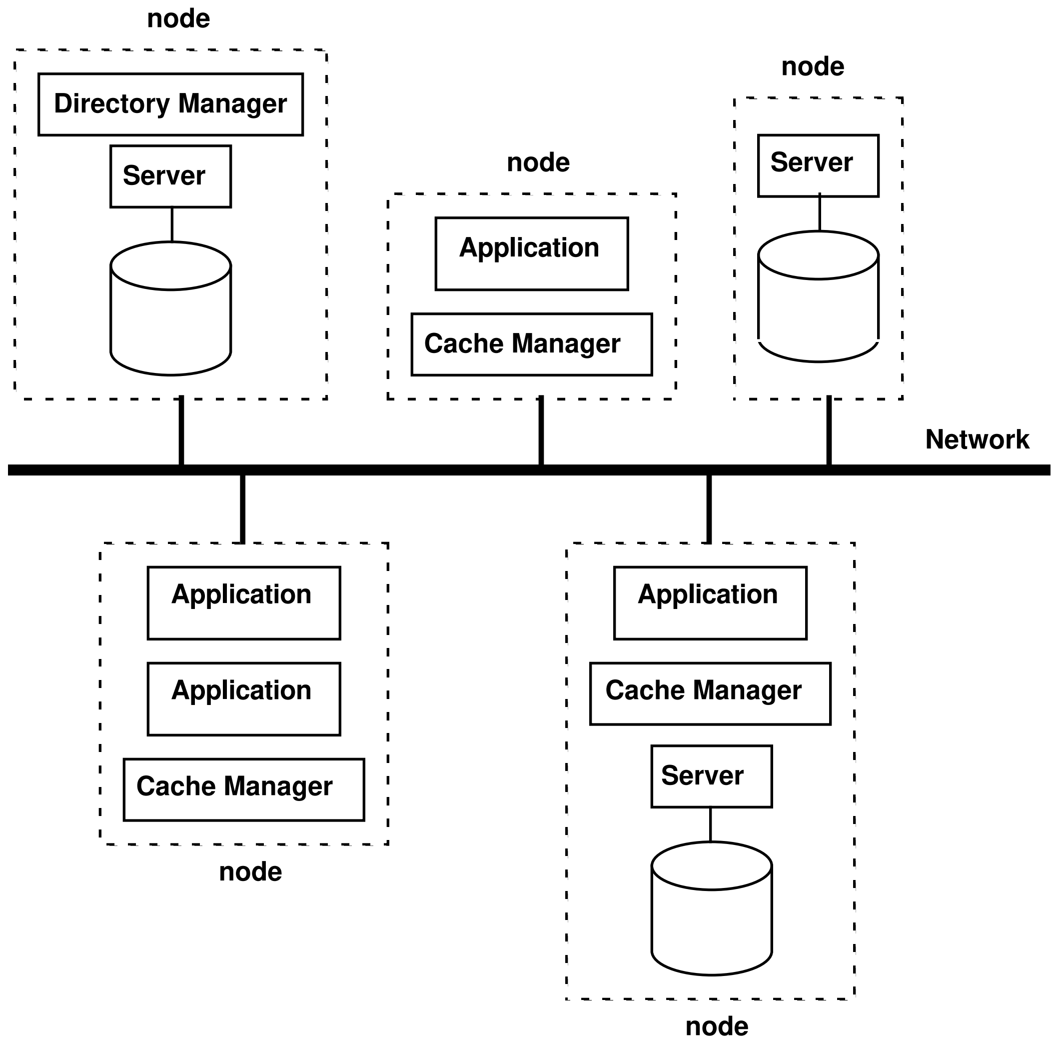


Figure 3.3 ObjectStore Process Architecture

ObjectStore executes a cache manager process on every client machine. The cache manager participates in the management of an application's *client cache*, a local holding area for data mapped or waiting to be mapped into the application's virtual memory. Only one cache manager is executed on a client machine, even if multiple clients are executed [Hal93]. The server and client communicate via local area network when they are running on different hosts, and by faster facilities such as shared memory and local sockets when they are running on the same host. ObjectStore's distributed architecture supports several network environments for interoperability among popular workstations and PC's, and includes support for TCP/IP, Novell IPX/SPX, and other popular network protocols.

Instead of storing databases as regular files managed by the operating system, databases may also be stored in special ObjectStore file systems managed by an ObjectStore *Directory Manager*. These file systems have a logical organization which is independent of that of any operating system file systems (and, in fact, may span several operating system file systems).

ObjectStore attempts to the greatest extent possible to present the application programmer with the appearance of a single object space consisting of both transient and persistent objects, rather than forcing the programmer to deal with separate application and database memory spaces, and explicit movement of data between them, as in relational database application programs. Thus, dereferencing pointers to persistent objects, cache coherency, etc. is handled entirely by the system, transparently to the programmer. Moreover, based on the requirements of CAD and similar applications, ObjectStore's architecture has as a primary goal to make the speed of dereferencing pointers to persistent objects be the same as that of dereferencing pointers to transient objects (once the persistent object has been retrieved from the database). That is, ObjectStore's goal is that there should be no additional overhead required to check whether the object pointed to by a pointer being dereferenced has been retrieved from the database already or not. To do this, ObjectStore takes advantage of operating system interfaces that allow virtual memory access violations to be handled by ordinary software (these interfaces are available in most standard versions of Unix, and other kernel-based operating systems such as OS/2). This "virtual memory mapping" approach is, so far, unique to ObjectStore among ODBMSs (and has been patented by ODI).

As described in [LLOW91], these virtual memory interfaces allow ObjectStore to set the protection for any page of virtual memory to *no access*, *read only*, or *read/write*. When an ObjectStore application dereferences a pointer whose target object has not been retrieved into the client (i.e., a page set to no access), the hardware detects an access violation, and the operating system passes this to ObjectStore as a memory fault. As noted above, ObjectStore maintains a client cache, consisting of a pool of database pages that have recently been used, in the virtual memory of the client host. When the application signals a memory fault, ObjectStore determines whether the page being accessed is in the client cache. If not, it asks the ObjectStore server to transmit the page to the client, and puts the page into the client cache. Once the page is in the client cache, ObjectStore calls the operating system to set the protection of the page to read only, to allow accesses to succeed. ObjectStore then returns from the memory fault, causing the dereference to restart and succeed. Subsequent reads to the same target object, or to other addresses on the same page, will run in a single instruction, without causing a fault. Writes to the target page will result in faults that cause the page access mode and lock to be upgraded to read-write. Since a page can reside in the client cache without being locked, some other client might modify the page in the database, invalidating the cached copy. A cache coherency mechanism involving the use of callback messages between clients is used to insure that transactions always see valid copies of pages. All virtual memory mapping and address space manipulation in the application is handled by the operating system under the direction of ObjectStore, using normal system calls.

ObjectStore databases can exceed the size of the virtual address space; also, two independent databases might each use the same addresses for their own objects. ObjectStore contains mechanisms for dealing with these problems. Specifically, ObjectStore dynamically assigns portions of address space to correspond to portions of the databases used by the application, and maintains a virtual address map that shows which database and which object within the database is represented by any address. As the application references more databases and more objects, additional address space is assigned, and the new objects are mapped into these new addresses. At the end of each transaction, the virtual address map is reset, and when the next transaction starts, new assignments are made. Each transaction is limited to accessing no more data than can fit into the virtual address space. In current applications, this limit is not often reached, since typical virtual address spaces are very large, and typical transactions are short enough so as not to access as many objects as would require this space. In other cases, the transaction could be broken into subtransactions and checked out into a separate workspace, isolated from other users.

Even though ObjectStore attempts to maintain a single object space abstraction for the application programmer, there are a number of aspects of the mapping that the programmer must be aware of. Specifically, the inter-object pointers that can appear in an ObjectStore program fall into different categories, and different validity considerations apply to these categories:

1. pointers from transient memory to transient memory
2. pointers from persistent memory in one database to persistent memory in the same database
3. pointers from persistent memory in one database to persistent memory in another database
4. pointers from transient memory to persistent memory
5. pointers from persistent memory to transient memory

Pointers in the first two categories are always valid. Pointers in the last three categories are valid only until the end of the current transaction. The validity restrictions on categories 4 and 5 are inherent in any mapping from a shared persistent object space (the database) to a private transient object space (the application program). The validity restriction for category 5 has to do with the particular way ObjectStore's handles multiple databases.

Generally speaking, an ODBMS's concurrency control mechanism guarantees that, within a transaction, copies of database objects mapped into the application's object space and the corresponding objects in the persistent database are synchronized. When the transaction ends, changes made to persistent objects by the application are written back into the database, and those objects are made available to other concurrently-executing applications. Pointers from persistent memory to transient memory become invalid outside a transaction because they are not usable by other applications accessing the shared database (they are local to the application), and the DBMS cannot guarantee that the object pointed to by the reference has not been moved by the client, deleted by the application, or even exists (the application may no longer be running). Pointers from transient memory to persistent memory become invalid outside a transaction because the persistent objects may move or be deleted by a concurrent application. (This is similar to the way in which, in a relational DBMS, there is no guarantee that rows of a database table that have been retrieved into an application program are the same as the rows actually in the database except within the scope of a transaction.)

ObjectStore provides means for overriding the restrictions on categories 3 and 4, guaranteeing the validity of these pointers. Specifically, ObjectStore can be directed to maintain the validity of pointers from transient to persistent memory across transactions. Similarly, ObjectStore can be directed to allow cross-database pointers in particular databases or segments. Alternatively, ObjectStore provides special enhanced pointers called *references* (class `os_Reference`, and its subclasses) that provide these facilities transparently, on the basis of individual pointers, at the cost of some additional dereferencing overhead and pointer space. ObjectStore references are similar in some respects to the ONTOS abstract references described in Section 3.2.

ObjectStore's clustering facility allows objects to be explicitly clustered to raise the probability that objects likely to be accessed from a given object are moved to application memory together, and thus are already in memory when referenced. For example, if all the components of a parts assembly are clustered together, then the first reference to one of the components causes the entire assembly to be cached in the client memory. Subsequent references to components are processed at memory speed. The programmer can specify that objects are to be allocated to a given database, to a specified segment of a database, or to a specific cluster with a segment. ObjectStore allows specification of a specific *fetch policy* with a given segment. This controls how much of the segment (in addition to the specific page the object is contained in) will be moved from the server to the client if an object in the segment is referenced, and the object is not already located at the client.

ObjectStore supports traditional database transactions employing two-phase locking, as well as additional forms of transactions required in complex applications such as "long" and "nested" transactions, as well as notification between users and other forms of cooperation among applications. ObjectStore also provides support for object versioning and configurations (see Section 6 for a discussion of these features), and supports full online backup for continuous processing environments. ODI has also indicated its intent to support multi-version concurrency control in a future (after Release 3.0) release of ObjectStore. This is a concept that allows read-only transactions to operate on a consistent version (effectively a snapshot) of the database, while other transactions modify the shared, latest version using conventional techniques.

ObjectStore provides an interactive graphical tool, SchemaDesigner, for defining object classes. This tool automatically generates C++ class definition header files corresponding to the defined classes. ObjectStore supports a metaobject protocol (MOP) providing run-time access to schema information[1]. The MOP, together with other ObjectStore facilities, provides for the easy use of third-party development tools with ObjectStore. ObjectStore also supports dynamic type creation for extending existing class definitions during program execution, and supports schema evolution for an application's metadata and existing object instances. [ODI94] describes a post-Release 3.0 product family called ObjectStore/DBConnect, designed to provide interfaces between ObjectStore and other database systems. Interfaces to be provided include:

- ObjectStore/SQL Gateway, which will support access to relational DBMSs including Oracle, DB2, and Sybase from ObjectStore.

- an ODBC interface to ObjectStore.

ObjectStore also provides database administration utilities for tasks such as:

- disk management
- performance monitoring
- backup and recover
- security and access control

ObjectStore supports associative (query) access to collections of objects, in addition to the "navigational" access using direct object references typically required by design applications. Queries treat persistent and non-persistent data in a uniform manner.

---

[1] It is not clear whether this MOP also supports the capability of changing various implementation or semantic aspects of the object model itself, as in the CLOS MOP [KRB91].

Indexes can be defined over collections of objects, and are used by ObjectStore's query optimizer to improve the efficiency of queries, as in a relational DBMS. However, in ObjectStore, indexes are not restricted to data members (attributes) of the members of a collection. Indexes may also be defined on *paths* (series of data members that span multiple objects) that may be frequently searched. For example, the statement

```
parts.add_index(pathof(Part*,designer->manager->name));
```

creates an index that uses as its key a path that goes from a part through the part's designer and the designer's manager to the manager's name. This could be used to optimize queries that involve finding parts designed by employees managed by particular managers.

ObjectStore's query facilities are based on the idea that database searching will be done primarily by navigation (following object references) from other persistent objects using object-valued properties, or by queries performed over explicitly-defined collection objects that play the role of type or class extensions (the set of all instances of a type or class). The system supports persistent variables (called *roots*) that can name either individual or collection objects, and can thus act as named entry points into databases.

ObjectStore supports navigation with standard C++ facilities, but also extends these facilities with associative query expressions that can be used within C++. This can be illustrated using some simple examples taken from [LLOW91].

The definitions below define two object classes (these would be specified in a C++ include file). Class `department` declares a data member (attribute) of type `os_Set(employee*)`, i.e., as a set of references to objects of type `employee`. `os_Set` is one of the ObjectStore-defined parameterized collection classes. If `d` is a department, then `d->add_employee(e)` adds `e` to `d`'s set of employees. `d->works_here(e)` returns true if `e` is contained in `d`'s set of employees, and false otherwise.

```
class employee
{
public:
   char* name;
   int salary;
};
class department
{
public:
   os_Set<employee*> employees;

   void add_employee(employee *e)
      { employees->insert(e); }

   int works_here(employee *e)
      { return employee->contains(e); }
};
```

ObjectStore includes a looping construct to iterate over sets. For example, the following code gives a 10% raise to each employee in department `d`. In the loop, `e` is bound to each element of the set `d->employees` in turn.

```
department* d;
...
foreach (employee*, d->employees)
   e->salary *= 1.1;
```

Unlike an embedded SQL, which has its own expressions that differ in syntax and semantics from those of the host (programming) language, ObjectStore queries are more closely integrated with the host language. A query is simply an expression that operates on one or more collections and produces a collection or a reference to an object.

C++ does not maintain class or type extensions, and neither does ObjectStore. Instead, as noted above, persistent sets of objects corresponding to type extensions (or relations in a relational database) must be explicitly declared and maintained. For example, in order to have queries that range over all employee objects in a database, the set of all employees must be explicitly defined, as in:

```
os_Set<employee*> all_employees;
```

This defines `all_employees` as a set of objects of type `employee`.

Selection predicates, which appear within query expressions, are also expressions, either C++ expressions, or queries. The following statement uses a query against `all_employees` to find employees earning over $100,000, and assign the result to `overpaid_employees`:

```
os_Set<employee*>&
overpaid_employees =
   all_employees [: salary >= 100000 :];
```

ObjectStore uses the notation `[: :]` to bracket queries. The contained expression is a selection predicate that is (conceptually) applied to each element of `all_employees` in turn. The query will be optimized if an index on `salary` is present, and the special brackets are intended to explicitly designate expressions that may be subject to optimization.

Any collection, even one resulting from an expression, can be queried. For example, the following query finds overpaid employees of department `d`:

```
d->employees
[:  salary >= 100000 :]
```

Query expressions can also be nested, to form more complex queries. The following query locates employees who work in the same department as Fred:

```
all_employees
[: dept->employees
    [: name == 'Fred' :] :];
```

Each member of all_employees has a department, `dept`, which has an embedded set of employees. The nested query is true for departments having at least one employee whose name is Fred.

[LLOW91] notes that most queries in ObjectStore tend to be made with selection predicates involving path expressions through objects and embedded collections. These paths become in effect materialized joins, and thus join optimization is less of a concern in ObjectStore than in relational DBMSs. However, ObjectStore's query facility also supports conventional joins. For example, if there is no stored connection between projects and engineers, the following query finds projects involving Fred:

```
projects[: engineers[: proj_id == works_on &&
                          name == 'Fred' :] :]
```

In this case, projects and engineers are matched by comparing the `proj_id` of a Project and the `works_on` field of an Engineer.

ObjectStore is among the most popular ODBMSs. A recent announcement indicated that a subset of ObjectStore will be the storage manager underlying the NeXTStep environment, and another subset of ObjectStore called the Persistent Storage Management Engine (PSME) will give SunSoft's Solaris DOE (Distributed Objects Everywhere) environment its object-storage capabilities [Var93]. Hewlett Packard also has rights to the technology for use in the HP/UX Distributed Object Management Facility (DOMF). In April 1993, IBM Corporation became one of the company's leading investors as part of a strategic relationship involving an equity investment, internal use, and joint development agreements.

3.4 VERSANT

VERSANT [Ver93a,b,c,d] is an ODBMS from Versant Object Technology designed to support multi-user, production applications in distributed environments. VERSANT runs on UNIX and PC platforms, including Sun, IBM, HP, DEC, Silicon Graphics, Sequent, as well as IBM-compatible PCs under OS/2. Windows and Windows NT versions are in beta testing. Versant emphasizes compatibility with evolving standards in their product, and has been an active member of the Object Database Management Group (ODMG).

3.4.1 Object Model

VERSANT uses a generic internal object model onto which multiple language interfaces are mapped. There is no special internal VERSANT database language. Essentially, the basic DBMS is accessible through the C interface, through which the internal object model functions can be called. The C++ interface provides C++ wrappers for those C functions. The internal object model is *passive*, in the sense that methods are not stored in the database, but rather reside in the applications that access the objects (see Section 6.3.1). As a result, the ODBMS need not deal with the problems of executing methods implemented in heterogeneous languages, but only with moving object state between the database and the accessing applications.

Each VERSANT object is assigned a *logical object identifier*, or *loid*. This identifier remains unchanged as long as an object exists in the database, and is not reused even after an object has been deleted. This permanent identification of each object is the basis for a number of VERSANT features, such as persistent links, cross-database references, and transparent object migration across databases.

VERSANT provides language specific interfaces for C, C++, and Smalltalk. Each such interface consists of several libraries of precompiled routines and, for typed languages, predefined data types. An interface can include its own development tools, and be used with other vendors' software development products.

When a class is created through a language interface, VERSANT creates a class object to store the definition in the database. For the C++ interface, VERSANT also creates for each class an associated runtime type identifier (class) object that contains additional information needed by C++. This runtime type identifier object is an instance of the C++/VERSANT class `PClass`.

Each persistent object and its class object are associated with a database. Persistent objects can be moved from one database to another, but they are always associated with a particular database. Because moving an object instance includes migration of its definition in a class object, a particular class object may be duplicated in numerous databases. However, once a class object has been migrated, it exists as an independent object in that database. This allows a migrated object to be used without being connected to its original source database. If classes of the same name are defined differently in different database, an attempt to migrate an object of a differently defined class will be automatically blocked.

The VERSANT C++ interface provides database functionality through three fundamental C++ object classes:

- `PDOM` (Persistent Distributed Object Manager), which contains methods for general database services, such as starting a session, connecting to a database, or committing a transaction

- `PClass` (Persistent Class), which supplies methods for classes, such as select (query the class extent), check out an object, or create an index

- `PObject` (Persistent Object), which provides general object functions, such as copy object, delete object, get object ID, as well as persistence services.

Any application-defined classes that require persistent behavior must be derived from the VERSANT-provided `PObject` class. These classes are defined in the application, along with their methods and ODBMS statements, compiled, and linked to the system libraries. The compile and link process in VERSANT includes the invocation (which can be turned on or off) of a Schema Capture utility called *sch2db* ("schema to database") that essentially does the same job as the ONTOS *classify* utility (creates database class definitions from classes defined in the program). The structure of a VERSANT C++ program is also similar to that of an ONTOS C++ program. All object access must be done within a session (essentially a short transaction; long transactions can span sessions). Objects to be updated must be marked as dirty, using the `dirty()` method defined by `PObject`. Such objects are automatically rewritten to the database when the enclosing transaction commits.

In the C++ interface, objects are made persistent by overloading the C++ `new` operation that creates the object, or by referring to the object from a persistent object. Once an initial persistent object has been retrieved from disk, subsequent object movement from disk to memory is transparent to the C++ program when it attempts to dereference an object pointer. An object can contain links to related objects. Each linked object is identified by its object identifier. VERSANT provides special C++ object reference classes, e.g., classes `Link` and `LinkVstr` (a variable-length array of object identifiers), to define these links (these are similar to ONTOS abstract references). Links can be defined as typed (can refer to only a specific class) and untyped (can reference any class). Bidirectional links can also be defined. Versant emphasizes the advantages of using these abstract object references, which ensure the safety of object references, over the use of direct pointers.

The VERSANT Smalltalk interface [Ver93b] provides database functionality through three fundamental Smalltalk object classes, which correspond roughly to the C++ classes described above:

- `ODBInterface`, which contains methods for general database services, such as starting a session, connecting to a database, or commiting a transaction

- `Class`, which supplies methods for classes, such as select (query the class extent), check out an object, or create an index

- `Object`, which provides general object functions, such as copy object, delete object, get object ID, as well as persistence services.

These are similar in many respect to facilities in the Gemstone Smalltalk interface.

Unlike C++, a Smalltalk image contains complete metadata that describes the structure and methods of object classes. The class information kept by Smalltalk is a superset of what is needed to describe a class to VERSANT. As a result, VERSANT class schemas need not be separately created for Smalltalk classes. When an instance of a class not already known to VERSANT is written to a database, the interface will automatically define the class to VERSANT. To make a Smalltalk object persistent in a VERSANT database, the object is sent the `becomePersistent` message, and it will be stored in the database at the next commit. Persistence is not determined at object creation time. Transient objects that are the target of a pointer in a persistent object also become persistent when a commit occurs. Because of this, care must be taken to ensure that only the desired objects become persistent.

Persistent database objects are automatically read locked and swapped into a Smalltalk image whenever they are sent a message from the Smalltalk interface. Thus, the interface is transparent to read only applications. Once a persistent object has been moved from a database to a Smalltalk image, it can be modified using normal Smalltalk mechanisms. To ensure that the database representation of the object is also modified to reflect these changes, the object must be sent the `vDirty` message (this corresponds to the `dirty()` method defined for `PObject` in the C++ interface). This marks the object for update, upgrades the current database lock to a write lock, and causes the updated object to be written to the database at the next commit. For example, the message `self vDirty` needs to be added to each method that modifies one of the object's instance variables.

3.4.2 Architectural Features

The VERSANT Scalable Distributed Architecture (VSDA) provides features such as transparent data distribution, object-level locking, dynamic schema evolution, and built-in workgroup computing support. Figure 3.5 shows the logical relationships that can exist between components of the VERSANT architecture. In VERSANT, the terms *client process* and *server process* refer to roles, not to specific hardware locations. The term *server* refers to a machine running *server processes* that support concurrent access by numerous users to one or many databases. The term *client* means an application process that can access multiple databases concurrently with other client applications.

The portion of VERSANT that runs on the same machine as a client application (typically a user's workstation) is called VERSANT Manager. VERSANT Manager implements the system's object model, and is responsible for maintaining a client object cache, coordinating distributed transactions (including 2-phase commit), object versioning, query processing, managing long transactions, schema management, checkout and checkin, and session management.

The portion of VERSANT that runs on the machine from which data is stored or retrieved is called VERSANT Server. VERSANT Server interfaces between VERSANT Managers and operating systems, and is responsible for page buffering, object locking, disk access, before- and after-imaging, transaction commit and recovery. There is a server process for each application database connection. The server process executes at the machine where the database resides. Multiple servers for a single database communicate with each other using shared memory.

VERSANT distributes work across the client and server. Object caching is performed on the client, which reduces network traffic and greatly speeds traversal of objects once the object has first been cached ("warm traversal"). Query processing is implemented in the server, which also reduces network traffic by only sending requested objects from server to client. Page caching is performed on the server, which reduces disk I/O for objects on frequently-used pages.

In the VERSANT architecture, a client and a server can both run on a single workstation, or clients and servers can be networked to allow access to multiple global databases. Each server can support multiple global databases, which are accessible by multiple clients. Each client can access multiple global databases on multiple servers in the network, and transparently access the objects managed by those servers. Objects may contain other objects on different nodes, and may be transparently migrated among nodes to improve locality of reference and performance without application code changes. Private databases, accessible to only one client, can also be created. A client can have a private database on its workstation, create private databases on a server, or use only global databases. The client and server component communicate via the VERSANT Network layer, which insulates them from the network protocol. If the client and server execute on the same machine, they may be linked to form a single process [EH92].

The C++ Interface consists of a C++ class library. and tools that allow C++ programs to access the Object Manager (including the C++ Schema Capture tool noted earlier), while the C Interface provides similar facilities for C programs. The Smalltalk interface was also described earlier. VERSANT View is a development tool that provides browsing facilities for database objects and classes.

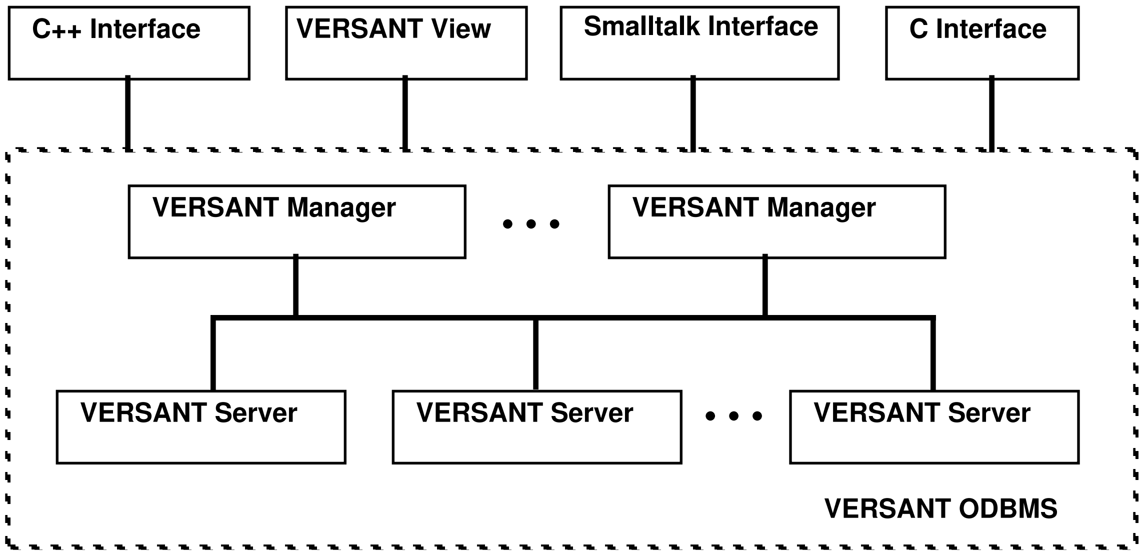


Figure 3.5 VERSANT Component Architecture

VERSANT provides numerous transaction options. First, transactions can be either short or long. A short transaction corresponds to a conventional DBMS transaction. A short transaction begins with an object request, and ends with a commit or abort. Distributed transactions involving objects located in multiple databases on multiple servers are automatically committed using a two-phase commit mechanism to ensure transaction integrity. A long transaction is a series of short transactions, and both its start and end points must be explicitly declared. During a long transaction, objects can be checked-out from a global to a private database. During the checkout process, various types of persistent locks can be set on the object. For example, an application can choose to be notified when an object is available, when a soft lock is preempted, when a new version of an object is created, or when an object is accessed, updated, or deleted. VERSANT also allows the definition of customized types of locks by defining symbols for user-defined lock modes and then specifying the relative precedence and compatibility of the various modes. A customized lock model must include the VERSANT standard lock modes as a subset.

Objects can also be specified as "versioned". When a versioned object is checked out, no persistent locks are placed on it, so that other users can simultaneously check it out for read or update. When a version is checked in, it becomes a child version of the original version. Concurrent write mode checkouts of the same object will result in parallel child versions of the same parent. The system tracks these versions automatically.

VERSANT also provides a C++ Application ToolSet, a set of application development tools integrated tightly with the VERSANT ODBMS. These tools include:

- VERSANT Screen--a graphical user interface builder which includes a C++-based scripting language (IC++) for application development

- VERSANT Report--a graphics-based report writer

- VERSANT Object SQL--an object-oriented SQL dialect which can be used within the IC++ scripting language, or embedded in C++ programs.

VERSANT Object SQL [Ver93c] provides a value-based means for locating selected objects in a database. These objects can then be used either as the basis for conventional object-oriented pointer navigation, or as the basis for further SQL queries.

VERSANT Object SQL statements are combinations of SQL-like and C++ statements. The structure of VERSANT Object SQL statements follows the structure of SQL statements. For example, database manipulation functions are implemented through statements such as SELECT, UPDATE, DELETE, and INSERT. However, the parameters that follow these keywords have a C++ structure. Common features of C++ such as pointers, class variables, and functions are used as VERSANT Object SQL parameters.

For example, in the statement

```
SELECT p.get_name()
FROM Person p;
```

SELECT and FROM are parts of the SQL SELECT statement, while the parameters that follow are C++. This statement selects all instances from the class Person and prints the value returned by invoking the method `get_name()` for each of those persons.

A VERSANT Object SQL SELECT statement extracts objects from the classes specified in the FROM clause and assembles their data into an array of tuples. For example,

```
SELECT
FROM Employee e;
```

returns an array of tuples, each containing one instance of the Employee class. If two or more classes are specified in the FROM clause, a SELECT statement returns n-tuples of objects from these classes as an array of tuples. For example,

```
SELECT
FROM Employee e, Department d;
```

returns an array of tuples, each containing an instance of Employee as the first element, and an instance of Department as its second element. In this case, the array will contain all possible pairs of Employee and Department objects.

Tuples and tuple arrays are themselves objects (a tuple array has type `OSQLResult`). These objects can be assigned to variables, and they or their elements processed by further statements.

Syntax is provided to allow users to indicate when searches of specified classes should also include subclasses of those classes. For example, the following query selects all employees whose weekly salary is greater than $750, including subclasses of employee:

```
SELECT FROM SUBCLASS
Employee *e WHERE
e->>get_salary() > 750;
```

(The "intelligent pointer" notation ->> instead of the standard -> or dot notation is required in a subclass search).

VERSANT Object SQL also supports various types of relational joins. For example, the following query selects employees and their departmental bonus multiplier:

```
SELECT e.get_name(), e.get_salary(), d.get_multiplier()
FROM Employee e, Department d
WHERE e.get_department().name == d.name;
```

The expression `e.get_department().name` in the WHERE clause is a *path expression* that indicates traversal of one or more object pointers.

VERSANT OSQL also includes versions of the UPDATE, INSERT, and DELETE SQL statements for modifying data.

A separate Objectworks\Smalltalk/SQL is provided as an interface between object DBMSs and the Oracle relational DBMS using the Smalltalk and SQL languages [Ver93b]. The interface is provided as a set of Smalltalk classes.

VERSANT supports runtime schema evolution. Corresponding database changes are made using a "lazy" approach: objects on disk may have an older 'storage class' and will be updated to the new schema when they are used. Schema evolution is permitted both on leaf classes (those with no subclasses) and superclasses. When one class is modified, changes are automatically propagated to child classes. VERSANT also provides runtime class creation and schema access.

Versant also provides the VERSANT Star and VERSANT Repository products. VERSANT Star is a platform for integrating object databases with traditional databases, applications, and file systems. It provides a set of gateways to integrate existing applications and databases. The VERSANT Repository is an object structure (defined as a class hierarchy) within the VERSANT ODBMS that stores the type and location of all objects in a distributed, heterogeneous database environment. This provides metadata information that can be used by applications at runtime.

Versant recently announced a new release (3.0) of VERSANT ODBMS which can store and manipulate objects generated from a variety of C++ compilers (some passive C++-based ODBMSs can handle only objects produced by a single C++ compiler). The new release addresses byte ordering incompatibilities and problems of the different compilers generating objects in different forms. The new release also provides full or incremental on-line backup. Versant has also announced that MCI Communications Corporation selected VERSANT Object Database Management System for an international network management application. An MCI team based in Colorado Springs is developing the application, which will be deployed at MCI's international network management center to handle functions such as event reporting. Versant has indicated that it sees network management, particularly in the telecommunications sector, as an increasingly important application for object databases (ComputerWorld, 5/2/94). It is said to be developing more fault-tolerant, network management, and event notification enhancements, and intends to support ODMG specifications by the first quarter of 1995. Versant and UniSQL are also jointly developing a SQL/X (the query language of UniSQL's UniSQL/X ODBMS; see Section 4.3) interface layer for the Versant ODBMS, which Versant will market, sell, and support ("Executive Brief", *Object Magazine*, 3(6), Feb. 1994, p.8).

[Hal93] cites the Visual Intelligence and Electronic Warfare Simulation (VIEWS) Workbench, a simulation system built using the VERSANT ODBMS [Woy93,94]. VIEWS was designed to enable analysts to build detailed intelligence and electronic warfare scenarios. The scenarios created by VIEWS are used to drive high-resolution intelligence and electronic warfare models. VIEWS was first implemented in C++ using a relational DBMS for persistent storage, and then modified to use VERSANT. The builders of VIEWS cited the following advantages of an object-oriented DBMS over the relational DBMS for their project:

- Better Schema Support: Use of the relational DBMS required that the C++ structures be flattened into relational tables. VERSANT directly captured the schema from the C++ application code. Also, it was noted that the translation between the relational tables to the internal C++ representation required extensive source code, which was unnecessary in the object-oriented DBMS.

- Better Application Language Interface: Use of the relational DBMS required developer knowledge of two languages: C++ and SQL. The object-oriented DBMS did not require developer knowledge of SQL.

- Better Application Performance: Reconstructing the complex objects in the C++ program from the normalized relational DBMS tables required complex joins between many tables which was expensive in terms of application performance. The direct representation of C++ objects and the client cache available in VERSANT provided improvements in performance of from 10 to 100 times over the relational DBMS.

- Additional Features: VERSANT provided additional features such as long transactions and versioning of objects which were not available in the relational DBMS.

[Woy94] noted that converting from the C++/relational DBMS implementation to the VERSANT implementation required:

- deriving all persistent classes from the VERSANT `PObject` class to inherit persistent behavior

- conversion of standard C pointers to the VERSANT `Link` class, the persistent database equivalent of a transient pointer

- adding the VERSANT `dirty()` method to all methods that update a persistent object

- converting transient linked list structures to the persistent VERSANT `VIList` linked list class

- removing C++ methods that explicitly implemented persistence in the persistent classes (since VERSANT handles this transparently).

3.5 ITASCA

ITASCA is a distributed active ODBMS developed by Itasca Systems, Inc. [Ita93, OFAQ93]. The ITASCA product is an extension and commercialization of the ORION prototype OODBMS developed at the Microelectronics and Computer Technology Corporation (MCC) [BKKK87; KBC+88; KBCG89; KGBW90,91]. Itasca Systems has added major enhancements and features, improved the performance, and strengthened the code, and continues to develop tools and other products to work with ITASCA. ITASCA runs on most Unix systems, including Sun, DEC, and IBM.

ITASCA has typical ODBMS features, including persistent storage for data and schema, concurrency control and locking, transaction management, multiple security levels, and logging and recovery for both CPU and disk media failure. Additional features of ITASCA include dynamic schema modification, long-duration transactions, shared and private databases, distributed version control, distributed transaction management, distributed query management, distributed change notification, object migration, and an extensible architecture.

3.5.1 Object Model

ITASCA implements its own object model, which supports the features generally found in most OODBMS models. Each object is assigned a unique object identifier (OID) along with a state and behavior. The OID encodes both the object's class and the private database where the instance was created. Attributes represent the state of an object, while methods (code) define the behavior of an object. A class collects objects that share the same set of attributes and methods. Subclasses derive from existing classes, supporting the definition of class hierarchies. Each subclass inherits all the attributes and methods of its superclasses. ITASCA also supports multiple inheritance. In ITASCA, classes, as well as instances, are first class objects. ITASCA also supports operator overloading (polymorphism), and run-time binding of methods to operations.

ITASCA supports composite objects built from component (child) objects. ITASCA maintains inverse links between these parent and child objects. Child objects can be exclusive to a single parent or shared by several parents. They may also be dependent or independent of their parent objects. If a child object is dependent on its parent, it is deleted when the parent is deleted.

ITASCA is an *active* database, meaning that the system stores and activates methods directly in the database. ITASCA is language neutral, in the sense that it implements its own complete object model, and does not rely on the application to provide part of the definition or implementation of objects. ITASCA objects are represented in a neutral format that is understood by each of its Application Programming Interfaces. Database objects stored using one programming language can be accessed or updated by other programming languages. Developers may write applications in C++, CLOS, C or Common Lisp [EH92]. Applications written in different languages may invoke the same ITASCA database methods. ITASCA database methods are written in Common Lisp (methods can also call existing C, FORTRAN, or Lisp code). In addition, applications may also use application-specific non-database methods or member functions.

3.5.2 Architectural Features

ITASCA is a fully distributed ODBMS, supporting both multiple clients and multiple servers. An ITASCA database contains both shared and private database partitions. The shared database contains objects that are accessible to all applications, even across sites. Private databases contain objects that are local to a given user or group of users. Because servers are distributed in ITASCA, each server has its own partition of the shared database. In addition, an ITASCA server can also have multiple private databases.

ITASCA stores each instance of data in one site. The system or a user may move the data from one site to another to improve data locality. Access to moved data remains transparent, so that there is no need for a user or application to know the specific location of data in the ITASCA distributed database.

ITASCA stores the schema redundantly at each site to improve performance. The schema also includes code in the form of methods. Management of schema updates is automatic for all sites. This includes sites that were off-line during any changes. Automatic distribution of schema changes, including method code changes, simplifies database administration.

No single site acts as a master site, thus ITASCA's architecture has no single point of failure. ITASCA has neither a central data server nor a central name server. This is important for maintaining a database system with high availability in a networked workstation environment.

ITASCA supports dynamic schema modification to create a flexible environment for changing or customizing a database system. Authorized users can add and remove attributes or change the subclass/superclass relationship at any time. Authorized users can also add or remove partitions of the shared database at any time. All this can be done interactively without affecting other parts of the ITASCA database at the time changes occur to the schema. There is no need to bring the system down or off-load/reload data to restructure the database.

ITASCA has a sophisticated security authorization technique tied to the class hierarchy. It supports both positive and negative authorizations at any level in the class hierarchy. For example, granting access to all objects but one requires only two authorizations: a global grant followed by a specific denial. Authorization extends to classes, instances of classes, attributes, and methods. Also, inheritance of authorization reduces the work of database administration.

Long-duration transactions allow users to check objects out of the shared, distributed database into their private databases. Users can then change the objects in the private databases without affecting the shared database or other users. These changes can be committed to the private database. Then, at any later time, the user can check the updated object or objects back into the shared database. Long transactions are persistent; the system can recover the state of a long transaction if a crash occurs while it is in progress.

ITASCA supports version control of objects. When a new version of an object is created, a version tree is formed. ITASCA also supports alternate versions such that multiple versions can have the same parent. Promoting an object version to a released status restricts any deletion of the object. ITASCA uses generic versions to dynamically reference the most recent or default version of an object without any intervention by a user or application.

Change notification in ITASCA is either flag-based or message-based. Flag-based notification is a passive notification scheme that will identify an updated object when the object is queried for such information. Message-based notification is an active notification scheme that will execute a method (or code) upon an update or other change to an object. Such methods can send mail messages or invoke other methods or programs.

Memory management in ITASCA uses both page and object buffers. ITASCA has a traditional database page buffer scheme that contains pages with multiple objects. Desired objects move from the page buffer to an object buffer. The object buffer then provides

ITASCA with enhanced in-memory performance because it contains only frequently-referenced objects. Object caching also occurs on the client side, which can improve the performance of applications.

The ITASCA long data manager manages multimedia data (linear and spatial data). Linear long data objects have a sequential internal format, such as text or audio. Spatial long data objects have two-dimensional internal format, such as a bit-mapped image. ITASCA provides built-in classes (which can be specialized) for audio, image, and text types.

The ITASCA architecture is also extensible. Many of ITASCA's kernel-level operations are defined as system methods. This allows them to be refined to customize their behavior. These methods include, among others, methods for making, changing, and deleting objects, as well as for checking-in and out, and making versions of objects.

The ITASCA executable includes a Lisp interpreter and compiler. A Local Lisp API allows Lisp application code to be loaded directly into the ITASCA executable image. A CLOS version of the Local Lisp API is also available. The Remote Lisp API allows a developer to access the database across a network form a base Lisp image. The C API provides access to objects in an ITASCA database through the use of a unique object identifier type. The C API may be used by other languages that permit calls to external C functions.

In the C++ API, a Dynamic Schema Editor (DSE) tool is used to define the database schema. The DSE automatically generates a C++ header file for those ITASCA classes that are to be accessible from C++. It is also possible to include prototypes for the ITASCA server methods residing in the database. These methods are called like ordinary C++ application methods, except that they are stored and execute on the server. The application then includes the generated header file and links against the ITASCA C++ API to be executable. The C++ API manages movement of accessed data member values from the database to the application as required.

ITASCA provides a query capability for DBMS applications. The query language performs queries over instances of classes. The scope of a query may be set by the user of the database to *private*, *shared*, or *global*. If the scope is private, only instances of the class in the current private database are examined during the query. If the scope is shared, then only instances of the class in the shared database are examined. If the scope is global, all instances of the class in both the shared database and the current private database are examined. An example of an ITASCA query in the C++ API is shown below [Hal93]. The query places all instances of the `Vehicle` class which have a `ManufacturerName` of `"Oldsmobile"` into the `obj_list` set. Note that the query expression is written in a Lisp-like notation, rather than C++. ITASCA also provides a query optimizer to optimize the evaluation of application queries.

```
ItascaSet obj_list;

Vehicle::select(
  obj_list,
  QUERY_EXPRESSION, "( equal ManufacturerName \"Oldmobile\",
  END_ARGS );
```

ITASCA/Ada is an API to the ITASCA database developed by EVB Software Engineering, Inc. It provides an object-oriented Ada interface that enables users to access ITASCA database objects from within an Ada application. As with other ITASCA APIs, objects accessed by Ada applications can be shared and accessed by applications written in other languages, including C, C++, Lisp, and CLOS. The design of ITASCA/Ada models the

ITASCA ODBMS as a set of seven classes, with each class represented by an Ada package, as follows:

- Package `ODBMS` provides support for session and transaction management, object clustering, and schema import/export facilities

- Package `Object` provides the basic Ada object model of the ITASCA database. Using an Ada limited private type for an object, ITASCA/Ada presents an object as an abstract entity that contains attributes and methods. Attributes and methods are themselves represented as first class, limited private Ada types and are encapsulated within corresponding packages described below.

- Package `Class_Attribute` encapsulates the Class_Attribute data type and corresponding operations.

- Package `Class_Method` encapsulates the Class_Method data type and corresponding operations.

- Package `Class_Object` supports the ITASCA class object (meta-class) concept.

- Package `Multimedia_Object` provides operations which pertain to multimedia and long data objects.

- Package `Versioned_Object` provides operations which apply to versionable objects.

ITASCA/Ada also provides a variety of generic Ada collections for manipulating structures like groups of objects, attribute lists, methods, and string lists. Collections of objects and types in ITASCA/Ada are implemented using the GRACE (Generic Reusable Ada Components for Engineering) components, a set of over 275 reusable Ada data structures. ITASCA database queries in Ada applications are submitted using the same query language as other ITASCA APIs. The objects returned from an ITASCA/Ada query are returned as a strongly typed, traversable Ada collection set which is implemented using the GRACE components. ITASCA/Ada is also designed to be migrated quickly to Ada 9X.

ITASCA also provides a set of tools. The ITASCA Active Data Editor (ADE) is an OSF/Motif application that provides an interactive graphical interface to ITASCA databases. It allows creation, modification, deletion, and querying of database objects. The Dynamic Schema Editor (DSE) provides a graphical facility for creating and manipulating an ITASCA class hierarchy (schema). The DSE may also be used to define and modify database server method code. These server methods are written in a Lisp-based 4GL that can be interactively tested and compiled. The Database Administration Tool (DBA) provides a graphical facility for administering ITASCA databases. DBA provides facilities for administering user accounts, monitoring and adjusting database configurations, and managing database files and partitions.

3.6 Objectivity/DB

Objectivity/DB is an ODBMS from Objectivity, Inc. of Menlo Park, CA [Obj93, OFAQ93]. It supports a fully distributed architecture, with operations working transparently over a mixture of multiple databases, schemas, users, and computers, and over heterogeneous hardware, operating systems, and networks. Language interfaces include a C++ class library interface; a C function library; and the SQL++ query language. Numerous administrative and GUI tools provide both an interactive and programmatic interface, and a messaging backplane allows integration of third party tools. Objectivity is an active participant in ODMG.

Objectivity/DB is resold by Digital Equipment Corporation as DEC Object/DB, providing a multi-billion-dollar second source vendor. Over 50,000 end users are licensed in production use, with applications including real-time telecommunications, aerospace, defense, case, CAD/CAM, CIM, manufacturing, oil & gas, process control, transportation, multi-media, case, document management, financial analysis, and corporate information management. Platform support includes all Sun, all DEC (including VMS, alpha, OSF-1), HP/9000 series (both 68xxx and PA-RISC), IBM RS/6000, NCR 3300, SGI, Windows 3.1, and Windows NT.

3.6.1 Object Model

The Objectivity/DB object model is based on C++. Objectivity/DB also provides the usual class library containing classes for strings, dictionaries, relationship management, etc.; class libraries for particular application domains are also available. In the standard C++ API, a simple object is a C++ class (or C structure) with associated access methods. Complex objects can have data attributes that are scalars, structures, aggregates, and fixed and dynamic variable-sized arrays.

Objectivity/DB also provides some additional non-C++ modeling concepts. Objects can be linked together using dynamic *associations*. The association mechanism supports uni- and bi-directional relationships, one-to-one, one-to-many, and many-to-many. *Composite objects* can be constructed by logically grouping objects through associations. Any number of composite objects may be contained in composite objects, and a single object may participate in any number of composites. Operations such as `delete` and `lock` can be defined to operate on a composite object as if it were a single object. A higher-level Object Definition Language (ODL) is provided that allows declaration of these additional Objectivity/DB modeling concepts. These declarations result in automatically generated methods and declarations for both C++ and C.

The standard C++ API allows application programmers to work with standard compilers and debuggers without extra preprocessors, providing ODBMS capabilities via overloading C++ operators (e.g., new, ->), and declarations via Objectivity/DB-provided classes. These include class `ooObj`, the superclass of all persistent object classes, and special object reference classes similar to ONTOS abstract references. Objectivity particularly emphasizes the advantages of its use of abstract object references, which ensure the safety of object references, over the use of direct pointers.

Objectivity recently announced a collaboration with ParcPlace Systems that will result in a version of Objectivity/DB that supports VisualWorks (Smalltalk). The product is expected to be in beta test in 4Q94. The new product will have an ODMG-93 compliant Smalltalk interface, and will support garbage collection, automatic schema migration, and object evolution, as would be expected within a Smalltalk environment. It will also implement relationships and versioning, and persistence through reachability (i.e., an object is automatically made persistent if it is reachable from another persistent object). The product will also provide mechanisms for sharing objects across C++ and Smalltalk applications.

3.6.2 Architectural Features

Objectivity/DB supports a distributed client/server architecture. Objectivity/DB uses Sun's Network File System (NFS), together with Objectivity/DB client software (object manager) and a lock manager to distribute the object database among multiple servers [ES94]. If the server is a Unix system, Objectivity/DB uses the NFS server but replaces the normal NFS client with its own. The Objectivity/DB client does most of the work in the architecture. It communicates with the file server and the lock manager to present a coherent view of the database to the programming interface. Any platform that supports NFS can be a database server. The lock manager, which coordinates concurrent access to shared objects, supports page level locking, and can reside on any machine in the network. A type manager component manages descriptions of all classes defined in the database.

Objectivity/DB/s storage hierarchy has four storage entities: *objects*, *containers*, *databases*, and *federated databases*. The base of the hierarchy is the object. Objects can be created in C++ or in SQL. Objects have a 64-bit OID and are stored in pages contained in files. A container is a collection of pages. The container is the basic unit of locking and clustering. When a program accesses an object, the client first checks to see if the page containing the object is in the client's local cache. If the page is not in the cache, the client accesses the server to read the page into memory. Objects that are closely associated can be clustered on the same page to improve performance. A database consists of a default container and a collection of user containers. The default container holds system objects. A database corresponds to a file, and is only limited in size by the operating system's ability to handle large files. A federation is a collection of databases, which can be distributed on multiple servers on a network. An object's 64-bit OID is a combination of four 16-bit integers that correspond to the database, container, page, and object numbers.

The architecture provides a single logical view over multiple databases on heterogeneous machines. Operations work transparently across this environment, including atomic transactions with two-phase commit, propagating methods, and versioning. Objects may be moved between databases and platforms without affecting working applications or requiring changes to applications. Multiple schemas may be created without affecting other users or databases, and may be used simultaneously with shared schemas, allowing local groups to define their own models but still connect to other groups. Databases may be detached from this shared environment (federated database) and used on portable devices, reconnected or moved to different (compatible) environment, or distributed as parts or image libraries. Gateways to RDBMSs (including Oracle and Sybase) are provided via third-party integration with Persistence Software using C++ class libraries (see Section 4.4). Persistence Software also provides access to any foreign data store, as long as the user installs the appropriate access methods, extending the single-logical-view to include read/write access to arbitrary foreign data stores. Together, these facilities allow delegation of responsibilities to the appropriate users, integration with existing systems, and gradual migration toward full enterprise-wide sharing.

Objectivity/DB stores objects in a platform-independent manner in the database. When objects are read into a local application, they are automatically converted into the appropriate local data type representation. Objectivity/DB automatically allocates enough storage in the database to store the worst-case representation of the object.

Objectivity/DB supports both short and long transactions. Short transactions are based on traditional (transient) locks, owned by the process, and group together an arbitrary set of operations. Long transactions are based on persistent locks, owned by the user, and provide the same arbitrary grouping. Default concurrency is two-phase locking and serialization, but extensions available include MROW (multiple-readers concurrent with one-writer), and allow users to lock with or without wait or with timed waits, to implement more sophisticated mechanisms.

Versioning is supported for individual objects, may be turned on or off at any time for each object, and may be restricted to linear versioning or allow branching with multiple writers. References to versioned objects may be to a specific version or to a version specified as the default version. Object configurations are supported through the association mechanism applied to versioned objects. When a new version of an object is created, the structure of the configurations in which it participates can be controlled by automatically dropping, moving, or copying its associations (see Section 6.5 for a further discussion of versioning and configurations).

Schema and object evolution are supported via versioning of type-defining objects. Each time a type definition is changed, its defining object is versioned, allowing arbitrary changes. Objects may then be instances of the old or new type version. Object evolution or upgrading to the new type version must be done by installing user-written conversion methods which can then be invoked by the system.

Numerous administrative and developer tools are provided, each with both an interactive and programmatic interface. These include graphical object and type browsers, query browsers, report generator, tools to examine and force short and long locks, to move objects and databases, etc. On-line incremental backup provides a consistent network-wide snapshot, including referential integrity across all databases, and runs incremental and full database backups with no need to quiesce the databases and no interference with active applications. All tools are built around a messaging backplane, which supports four levels of integration with user and third-party tools. Integrated products include HP SoftBench, CenterLine's ObjectCenter, the Persistence RDBMS gateway, PTech and ProtoSoft Design and Analysis, and XVT and UIM/X.

The SQL++ product provides support for ANSI SQL, plus additional object extensions. Query predicates may use either C++ or SQL syntax. The object extensions include the ability to call methods in query expressions, traverse object relationships, access to inheritance hierarchies, and the ability to query nested substructures. SQL++ also supports the ODBC and SQL Access Group (SAG) protocols. Queries may be invoked programatically or interactively.

Objectivity/DB also provides an interface to the SunSoft DOMF (Distributed Object Management Facility), SunSoft's implementation of the OMG CORBA architecture. This interface allows direct access to objects in Objectivity/DB through the CORBA architecture.

3.7 O_2

O_2 [BBBD+88, BDK92, LRV88, Deu91] is an ODBMS from O_2 Technology, a French company (it is distributed in the United States by Scientific Services, Inc.). The system is based on research work supported by the Altair research consortium. In addition to providing O_2Engine, an object database engine, O_2 provides programming language interfaces (e.g., to C and C++), an object query language, a user interface generator, and a graphic programming environment including a debugger and a schema and database browser. O_2 is available on major Unix platforms. O_2 Technology has been an active participant in the ODMG, and in particular is the source of many of the object model and query language concepts in the ODMG specifications.

3.7.1 Object Model

The O2 object model is interesting from the point of view of merging relational and object technology in that it includes both *objects* in the conventional object-oriented sense, and structured *values*. Values may contain objects, and objects may contain values, and these structures may be deeply nested as needed. However, values differ from objects in that:

- the structures of values, while similar to those of objects, are not encapsulated
- values have no identity
- values are manipulated by operators, not by methods

Operators are predefined for *tuple*, *set*, and *list* values. As a result, in the O2 model, tuple (row) values, for example, behave exactly like tuples in the relational model, except that tuples can contain objects, and other values, in columns. Example (non-object) type declarations for types representing Documents and Sections of documents in the O2 model might be:

```
type Document: tuple (title: string,
                      printdate:  date,
                      author: tuple(firstname: string,
                                    middlename: string,
                                    lastname: string),
                      components: set(Section));

type Section:  tuple (title: string,
                      number: integer,
                      components: set(Paragraph));
```

In this declaration, type `Document` is defined as a tuple, rather than an object. The `author` property, rather than being defined (perhaps) as a reference to an object of type `Person`, is instead defined as a nested tuple. The `components` property is defined as a nested set of tuples of another tuple type, type `Section`. Structured values such as tuples, lists, etc. can play the role of the corresponding object type constructors used in other ODBMSs. All values have types which are recursively definable from pre-defined atomic types and type constructors such as `tuple`.

In O2, objects have an *identity*, a *value*, and a behavior defined by its *methods*. An object belongs to a *class*, rather than a type. An example of a class definition might be [Deu91]:

```
class City
   type tuple (name: string,
               map:  Bitmap,
               hotels: set(Hotel)

   method how_many_vacancies(star: integer): integer,
          build_new_hotel(h: Hotel)
end;

class Hotel
   type tuple (name: string,
```

```
                 read stars: integer,
                 read free_rooms: integer)

    method reserve_room: boolean,
           check_out
end;
```

An object or a value may refer to other objects via their identities. Similarly, an object or a value may be composed of subvalues. However, a subvalue is part of its containing object or value, and cannot be shared with other objects, while a referenced object is independent of its containing object or value, and can be shared. Object encapsulation is supported. Attributes are private to a class by default, unless explicitly made public by a `read`-only or `public` specification. Methods may also be private or public.

The implementation of a method is separated from its specification inside a class. The body of a method of the City class specified above is shown below. It is written in O_2C, but could be implemented in C or C++ as well.

```
method body how_many_vacancies(star: integer) in class City {
          int number = 0;
          o2 Hotel h;
          for (h in self->hotels where h->free_rooms > 0 &&
               h->stars == star)
               number += h->free_rooms;
          return number;
}
```

A class can inherit its type and methods from other classes. Attribute or method name collisions are resolved by explicit renaming in the subclass. An inherited type or method can be redefined locally, provided that the redefinition satisfies certain rules to preserve subtyping semantics (substitutability).

Objects or values become persistent by being attached directly or transitively to a persistent "root". These roots are declared in the database schema by giving a name to them. This *persistence model* is referred to as *persistence by reachability*, and has the following characteristics:

- When an object is made persistent, so are its component objects (transitively). This frees the user from having to perform this task explicitly.
- Unreferenced objects are automatically garbage-collected.
- The user is not required to decide whether an object will be persistent when creating it. Instead, objects can be made persistent after they are created, when the need becomes clear.

Section 6.1 discusses this and other ODBMS persistence models.

3.7.2 Architectural Features

The general architecture of O_2 [Deu91] is shown in Figure 3.7.

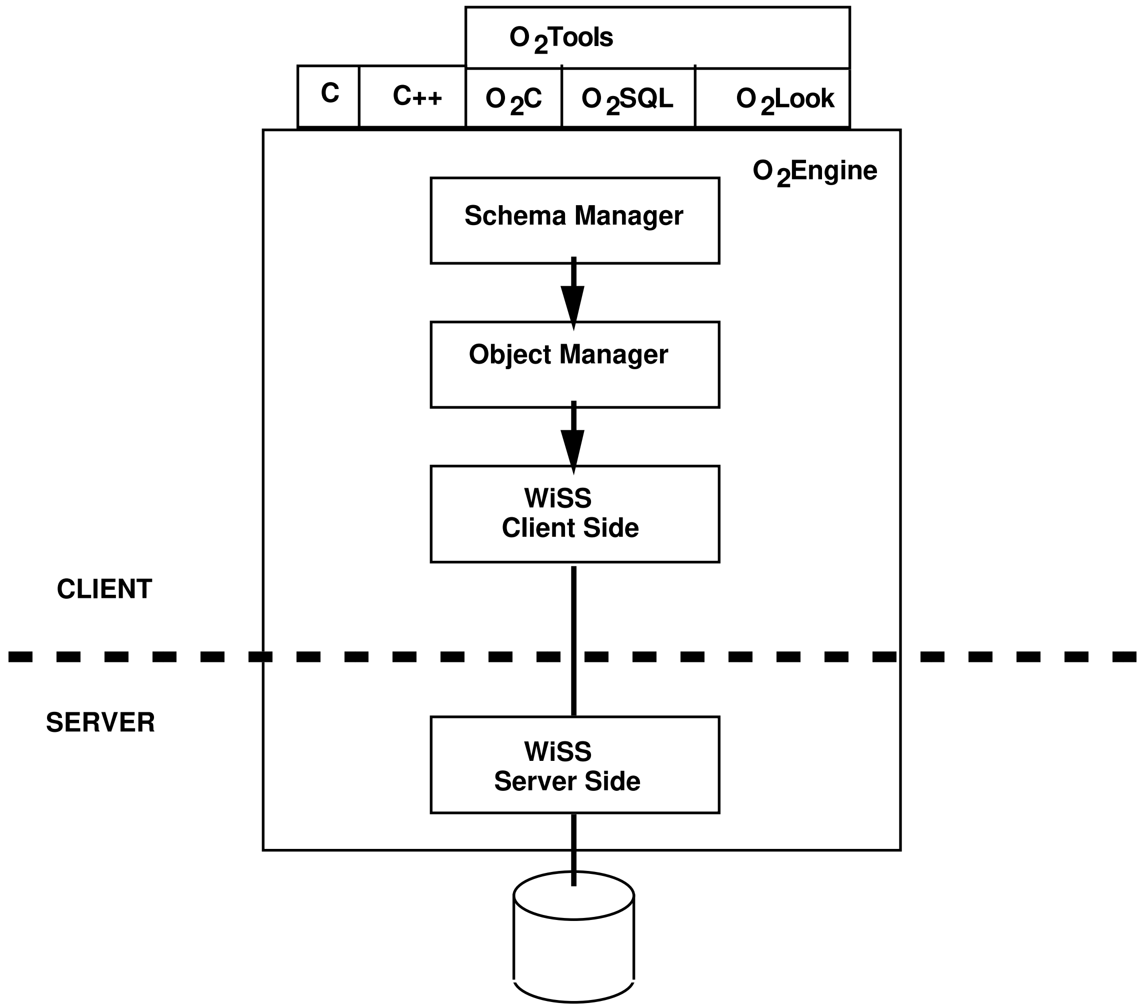


Figure 3.7 O_2 Client/Server Architecture

O_2 includes the following components:

- O_2Engine, a language-independent object database engine supporting the object data model.
- O_2Look, a graphical tool for creating, browsing, and editing database objects (an extension, O_2Graph, is also available).
- O_2SQL, an SQL-like object query language which includes a query optimizer and support for method invocation.
- O_2Tools, a complete programming environment including graphical browsers and editors, a symbolic debugger and source-level manager, and integrated schema documentation.
- C and C++ interfaces
- O_2C, a 4th generation programming language (a superset of C)

An interface between O_2 and Lisp is also available.

As described in [Deu91], O_2Engine is composed of three main layers. The upper layer of the system is the Schema Manager. It is responsible for the creation, retrieval, update, and deletion of classes, methods, and global names. It is also responsible for handling the semantics of inheritance and for checking schema consistency. The Object Manager is the middle layer. It handles objects with identity and passes messages, manages structured values and their structured operations. It also implements the reachability model of persistence, garbage collection, and implements indexes and clustering strategies. The lowest layer is an extended version of the Wisconsin Storage System (WiSS) [CDKK85], which serves as the O_2 disk manager. WiSS provides record-structured sequential files, B-tree and hashed indexes, and long data items as persistent structures. These structures are mapped into pages, which are the basic persistence unit. WiSS provides indexes for disk management, as well as full control of the physical location of pages on disk. WiSS implements a conventional "flat" (unnested) transaction model. Locking is used for concurrency control, and a write-ahead log technique is used for rollbacks and recovery.

O_2 implements the form of client/server architecture called a *page server*, in which the server deals only with pages, and does not understand the semantics of objects (see Section 6.2 for a further discussion of ODBMS architectural variants). 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 majority of the available CPU cycles are expected to be available, leaving the server to perform the tasks that it alone can perform. 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 4K bytes is not much higher than the cost of sending 100 bytes. 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. Finally, by minimizing the load each workstation places on the server, it is possible to support more workstations using the same server.

Because the server does not understand the concept of object, O_2Engine cannot issue queries or execute methods on the server. However, indexed access for large collections is still possible from the client, since the index code is built on top of the client's page-cache level. O_2 Technology's experience with an earlier object server architecture was that a significant overhead was imposed on the system to address potential cache inconsistencies (e.g., an updated version of an object may exist in the client's cache but not in the server's cache) that may arise in that architecture.

The client/server split is made at the WiSS level, which deals with pages. WiSS is split into a workstation (client) process and a server process in the following way: The server process consists of (i) a layer providing the storage and retrieval of disk pages plus disk resource (pages, extents, files) allocation, (ii) a page-cache layer, and (iii) concurrency control and recovery services. The server is accessed by an RPC interface. The client process consists of the page-cache level plus all the higher levels. In addition, a local lock manager caches locks acquired by the client on the workstation. Calls made to the I/O level of WiSS by the WiSS software running on the workstation are turned into RPC calls to the server. However, these calls are not turned directly into I/O calls to the server's disk drives. Rather, since the server also maintains a page cache, they become calls to the server cache.

Objects and values are represented in O2Engine in terms of records (and other data structures for large collections). The format of objects is identical in memory and on disk, avoiding any conversion overhead. Record identifiers are used as persistent identifiers for objects and as pointers for constructed values. On disk, records storing objects or values refer to each other by physical identifiers, i.e., identifiers reflecting disk locations. WiSS record identifiers are used directly. These identifiers consist of a 2-byte volume identifier, a 4-byte page identifier within a volume, and a 2-byte slot number. O2Engine uses a two-level addressing mechanism: persistent record identifiers on disk, and handles for records in memory. Each handle denotes a distinct record containing a value or an object. When getting a component object y of a persistent object x, the system dereferences the persistent record identifier (PID) of y by using a PID-to-handle hash table. If an entry exists for y the handle will be returned to the caller, otherwise a handle is created for it and the corresponding entry is inserted into the table. Pointers are not swizzled (converted from PID to handle) directly within records because the unswizzling operation would have to be performed when sending a dirty page back to the server.

The recovery algorithm is log-based, and involves both the client and the workstation. The log is handled by the server and is essentially a redo log. Concurrency control is handled with a hierarchical, two-phase locking protocol on files and pages. The global lock manager runs on the server, and a local lock manager caching the set of granted locks runs on the workstation.

O2 supports a flexible C++ interface. A user can either export O2 classes to C++, or import C++ classes to O2. The *export to C++* command automatically generates C++ classes from O2 classes. The user can choose to export the type of a class as well. If this is done, the generated class automatically includes C++ methods to read and write C++ objects to and from the O2 database. The read method is transparently called when accessing the object for the first time, while the write method is transparently called when it is necessary to write the object back to the database (e.g., when a transaction commits). After the initial read, the object is accessed directly, as with any C++ object. Each generated C++ class X comes with a twin class O2_X, which acts as a persistent pointer to objects of class X. The C++ program must manipulate persistent objects through this pointer. This means that instead of declaring a pointer to a class X *p, the pointer must be declared as O2_X *p. Alternatively, C++ classes can be imported into O2. In this case, the *import from C++* utility is used. This utility generates the necessary O2 declarations, as well as the read and write methods, and the corresponding pointer class, as described above. The user must then adjust the C++ declarations so that they use the proper persistent pointers, instead of ordinary C++ pointers. O2 also provides a C++ class library to enable C and C++ programs to access the other facilities of O2.

O2 also supports O2SQL [Deu91, BCD89], an SQL-like query language. O2SQL generalizes standard SQL by including, for example:

- capabilities for processing both values (including complex values) and objects, such as path expressions to reference embedded structures or follow object references
- invocation of user-defined operators in queries
- user-defined collections, including both sets and lists, to be referenced in `from` clauses

O_2SQL is defined as a subset of O_2C, but can be used independently as an ad hoc interactive query language or as a function callable from C or C++. As noted above, O_2SQL has heavily influenced the query facilities in the ODMG specifications [Cat94a].

As a simple example, a query to find the parts of a user's car (designated by the name `MyCar`) costing more than $50 would, in O_2SQL, be:

```
select x.name
from x in MyCar.components
where x.cashvalue > 50
```

Of particular interest in O_2SQL are the facilities provided for accessing and constructing structures. These are somewhat more complete than those found in query languages for many Object DBMS products. For example, given the `Document` and `Section` types defined earlier, the query to find the titles of the sections with numbers less than 5 in documents printed on a certain date would be:

```
select x.title
from x in flatten select y.components
                  from y in Document
                  where y.printdate = "01/04/78"
where x.number < 5
```

The nested query returns a set containing as many sets of sections as there are documents printed on the given date. The `flatten` operation returns a set of sections that is the union of these sets of sections. This set is then queried in the outer query. [BCD89] indicates that a simpler syntax which takes into account logical dependencies between query variables could also be used, namely:

```
select x.title
from y in Document, x in y.components
where y.printdate = "01/04/78" and x.number < 5
```

Nested structures can also be constructed in O_2SQL expressions. For example, based on the `Document` example, the following query constructs a tuple whose `components` property is a set of tuples.

```
tuple(
  title: MyDoc.title,
  components: select tuple (title: x.title, number: x.number)
              from x in MyDoc.components
              where x.number < 5)
     )
```

In this case, the set value of the `components` property is constructed by a query whose select clause contains a tuple constructor. The ability to flexibly query and construct such structures is highly important if a query language is to manipulate complex data types.

The O_2 user interface generator, O_2Look, supports the display and manipulation of large, complex, and multimedia objects on the screen. O_2Look provides facilities for creating graphical user interfaces quickly, and provides a set of predefined widgets for constructing displays. O_2Look also provides high-level graphic facilities to display, edit, and browse through database objects.

O_2Tools is a graphical programming environment supporting the development of O_2 applications. It allows the programmer to browse, edit, and query the database and the schema, and to edit, test, and debug methods and programs.

A considerable amount has been published on various aspects of the formal underpinnings of the O_2 model and its query facilities [AB88, AK89, CDLR89, LRV88]. Such analysis is essential if powerful query facilities corresponding to those in relational DBMSs are to be provided for Object DBMSs. According to Internet postings from O_2 Technology staff, the future version of the O_2 C++ API will be ODMG 93 compliant. O_2SQL is also ODMG 93 compliant.

O_2 is reported to have been used to implement a network administration platform at France Telecom, which defines within a common kernel a set of generic O_2 classes conforming to the GDMO standard and necessary for defining specific applications. The platform is used, for example, to develop applications for managing customer networks.

3.8 MATISSE

MATISSE[1] [Int93, Sut93] is a second-generation ODBMS from ADB (formerly ODB/Intellitic). The first-generation product, G-BASE, was described in [Man89a]. MATISSE is a multi-user client/server system, and is available on Sun and VAX/VMS platforms. MATISSE supports a highly adaptable object model, which can be extended by the user. The MATISSE API is a set of C library functions that can be bound into any language that supports an external C function call. The API supports all features of the database. MATISSE also provides an interactive Object Editor, which allows direct access to the database content through either text-based or graphical interfaces.

3.8.1 Object Model

MATISSE implements an object model that is independent of any programming language. This object model is built as an explicit layer on top of a more primitive *micromodel*. Specifically, MATISSE supports the following descriptive layers [Hal93]:

- micromodel: The *micromodel* is the lowest level of the MATISSE database. It implements a very simple data model of objects and connections. The micromodel is used to build templates.

- templates or metamodels: A *template* or *metamodel* is a schema defined using the micromodel which describes a specific data or object model, i.e., the rules to follow when designing a database schema. The only metamodel which currently exists for MATISSE is an object-oriented metamodel. This metamodel is used by the object-oriented services API. MATISSE states that metamodels for other data/object models could be developed on top of the micromodel. For example, a relational data model could be developed for MATISSE.

- schema: A *schema* is a user-defined set of data/object descriptions which is used by an application.

---

[1] Recent information suggests that the product is now named "M.A.T.I.S.S.E." (i.e., with periods after each letter). This section retains the older name.

The MATISSE micromodel is based on the semantic network concept, with enhancements to support relationships, values, entry points (indexes), and triggers (which generate calls to functions or procedures when relationships or values are modified). The MATISSE object-oriented metamodel is a self-descriptive semantic network that defines the concepts of classes, relationships, attributes, messages, and methods (see Figure 3.8 [Sut93]). All MATISSE concepts are first class objects including instances, classes, and the concepts defining the MATISSE metamodel (all are stored as objects in the database).

All objects in MATISSE have a unique immutable identifier and belong to a single class (an object can be asked its class at runtime). In MATISSE, as in many object systems, a class is an implementation of a type. The interface to a class consists of methods (or functions) attached to the class and messages which are used to invoke the methods. Attributes and relationships may also be defined for classes.

Methods are independent objects and can be written in arbitrary languages. In addition to methods associated with specific classes, MATISSE also provides *universal methods* which apply to the entire database, and are not attached to a particular object class. Methods may be shared by multiple classes, and can be queried to determine the classes to which they apply.

Complex structures can be defined either by embedding literal values in MATISSE objects or as OID-based relationships. For relationships, linked objects are not considered "contained" in the sense that deletion and other actions are not automatically propagated unless the user specifies an appropriate trigger. MATISSE includes built-in aggregates such as lists and arrays. Additional types of aggregate can be supported, but the user must specify the behavior by supplying the appropriate links and triggers.

Classes, including class `Class` and its associated objects, are first class objects in MATISSE, i.e., each method is stored as an instance of class `Method`, each attribute is stored as an instance of class `Attribute`, and so on. They are physically stored as objects on the server, and may be modified in the same way that other objects are modified. This allows support of on-line schema evolution. MATISSE allows any attribute of any object to serve as an "entry point" or index.

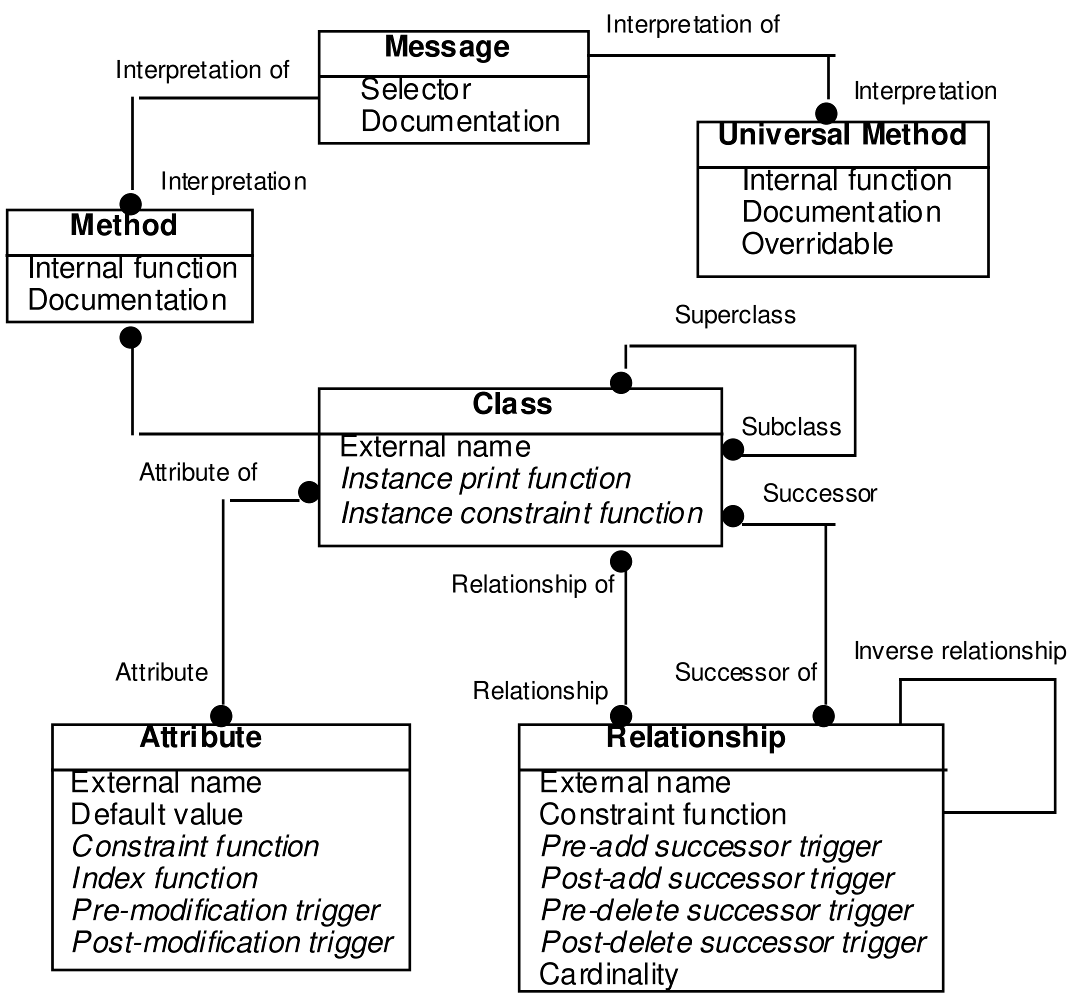
Class `Relationship` supports specification of the cardinality of a relationship. It also includes user-specified triggers that fire before and after adding or deleting a link. Inverse relationships are automatically supported. Referential integrity is guaranteed by enforcing the rule that no object can be deleted until its links to other objects are deleted. A variety of constraints and/or triggers can also be defined for attributes and classes as well, allowing complex user constraints to be defined and enforced.

MATISSE supports standard "specialization" inheritance. Subclasses consist of all attributes, relationships, messages, and methods of the superclasses, plus those additionally specified by the user. Multiple inheritance is supported and all instances of a subclass are considered instances of the superclasses. MATISSE does not allow name conflicts for inherited attributes and relationships in multiple inheritance, although methods may be overloaded. In multiple inheritance, invocation of methods occurs in a left first, depth first manner determined by the way the user specifies the class hierarchy.

In addition to inheritance, forms of delegation can be implemented by providing a method, trigger, or constraint that invokes other methods or functions. In addition, universal methods, which are independent of the inheritance hierarchy, can be invoked by any method or function attached to the database, or from the application through the MATISSE API.

The structure and behavior of MATISSE objects is defined by the MATISSE metamodel. The user can modify the metamodel, which effectively changes the MATISSE object model. The metamodel can also be modified to support specialized data models and functionality. For example, the metaclass `Attribute` can be modified to allow read and write to data in an external relational database. This allows any attribute of any object in the system to be specified as residing in an external database. When an object is accessed, such attributes appear as part of the object transparently to the user.

Schema evolution is similarly supported, with the constraint that database consistency can be enforced by the object manager. All extensibility is dynamic, and can be accomplished while the database is live. The object manager will enforce consistency of data in the database, and provide appropriate error messages if the user tries to make changes that will render data inconsistent. Security restrictions can be applied to control who is allowed change schemas (or the metaschema).



Note: Attributes in italics are user definable

Figure 3.8 MATISSE Object MetaModel

MATISSE makes no attempt to provide close ties with any particular programming language. The MATISSE Application Programming Interface (API) is a set of C function calls providing access to the facilities of the MATISSE object model. If a programming language (such as C++) supports an object model, then the programmer must deal with both the language's object model and the MATISSE object model. The MATISSE class hierarchy is similarly independent of any language class hierarchy. Typically, a user develops an application hierarchy that consists of persistent and non-persistent objects. The persistent objects would be created in the MATISSE database by calls to the MATISSE API, with the application programmer handling the translation of objects between the language and the database.

3.8.2 Architectural Features

MATISSE uses a client-server architecture having the following components:

- a runtime Server Engine which supports historical versioning, multi-threading, and transparent data replication
- Server Engine libraries
- a client library linked to the application (there is no separate client runtime)
- an Object Editor, which allows object creation and manipulation through a GUI interface
- an Object Browser
- DB Administrator tools

The micromodel and metamodel semantics are handled by the client library. The Server Engine is responsible only for data persistence and concurrency control. The client library is accessible through a C language API. This API provides all the functions to perform database operations such as opening transactions, creating and modifying instances and classes, and navigating relationships. The Server Engine uses a multi-threaded architecture, and can take advantage of asynchronous and parallel disk I/O subsystems when combined with intelligent controllers. The Server Engine also uses various disk allocation strategies (e.g., automatic clustering of small objects) to improve disk performance. The client and server communicate through RPC mechanisms; all objects are stored on disk by the server, but cached in client memory for use by client applications.

Concurrency is supported by an implementation of both two-phase locking and two-phase commit protocols. By default, locking occurs implicitly. However, the user may control the locking sequence and locking is not required for most reads, because historical versions are present in the database (see below). The client cache is flushed between transactions, except for objects which the user pins in client memory. This guarantees that the client always sees the current version of objects in a multiuser environment (except for pinned objects).

All objects (metamodel objects, class schema, and instances) are stored on disk in buckets, which are the discrete units of transfer of information from client to server. The bucket size can be optimized by the user to tune application performance. Schema and other objects are dynamically loaded into client memory as needed. Only those objects needed for current computations need to be sent over the network from server to client.

Objects stored on disk contain a reference to the methods and messages which invoke them. The object code of a method is stored on the client workstation and executes on the client (except for certain operations which occur on the server for performance optimization). This architecture provides for high volume transaction processing and allows for heterogeneous hardware environments.

MATISSE is a completely versioned database supporting copy semantics. A change to an object always results in creation of a new version of the object without altering previous versions or the object identifier (OID) of the object. Versioning is orthogonal to the object model, and is handled transparently by the MATISSE Server Engine. MATISSE maintains versions of all objects, including schemas. Users can see a consistent view of any historical state of the database.

The state of the database is determined by logical time, which is incremented at the commit time of each transaction. The database may be viewed from any logical time specified by the user. The user sees all objects which existed at the specified logical time in the state as it existed at that logical time. This includes metamodel objects and class schema objects, as well as instances.

MATISSE supports explicit object deletion rather than garbage collection. Version collection occurs at the discretion of the user. If all version collection is done immediately, the database operates in the same way as database systems which have no inherent knowledge of object history. Typically, version collection is done on a user-specified, periodic basis after execution of a "save time" function which flags a historical logical time as a view of the database that should not be collected. Reporting or on-line backup can be done without locking by executing reads on the "save time" view of the database. If reporting or backup is done at low priority, this has negligible effect on performance of production transaction processing. When historical versions are no longer needed, they may be version collected at the discretion of the user. MATISSE is designed to retain knowledge of all historical states of the database through systematic archival of version collected objects.

The next release of MATISSE intends to support ANSI SQL92 with object extensions. The evolving SQL3 specification will be fully supported as it stabilizes.

3.9 Other Object DBMSs and Related Developments

Previous sections have described a number of representative ODBMS products. However, there are other ODBMS commercial products, research activities, and related systems that are also of interest for various reasons. This section briefly describes several of these systems/activities. This section does not attempt to be exhaustive in covering such activities. Numerous other such activities, such as ENCORE [HZ87, SZ87, SZ89], are not described due to space and time constraints.

3.9.1 ODE

Ode is a public-domain ODBMS developed at AT&T Bell Laboratories. Ode runs on (at least) Sun (Sparc) workstations and users must have C++ release 2.0 or a later release.

Ode is available to universities at no charge (however, AT&T requires the signing of a non-disclosure agreement). It can be obtained via ftp from research.att.com [OFAQ93].

Ode is based on the C++ object model. The Ode database is defined, queried and manipulated in O++, an upward-compatible extension of C++ with database programming language facilities. O++ programs can be compiled with C++ programs, thus allowing the use of existing C++ code.

In the Ode model of persistence, memory is partitioned into volatile and persistent memory. Volatile objects are allocated in volatile memory. Persistent objects are allocated in persistent store and continue to exist after the program that created them has terminated. An Ode database is a collection of persistent objects. Each object is identified by a unique object id (a pointer to a persistent object).

O++ provides facilities for creating and manipulating the Ode database. For example, O++ provides facilities for specifying transactions, creating and manipulating persistent objects, querying the database, and creating and manipulating versions. In addition to O++, other interfaces have also been defined for Ode:

- OdeView, a graphical X-based interface to the Ode database.

- OdeFS, a file system interface to the Ode object database. OdeFS allows objects to be treated and manipulated like files. Standard Unix commands such as rm, cp, and mv, and tools such as vi and grep can be used to manipulate objects in the database.

- CQL++, a C++ variant of SQL for easing the transition from relational databases to object-oriented databases such as Ode.

EOS, the storage engine of Ode, is based on a client-server architecture. Some features of EOS include:

- Support for large objects, which are critical for multi-media applications. Ode provides both transparent access for large objects and a file-like interface for large objects. The latter can be used to access and update parts of a large object.

- Concurrency control, which is based on multi-granularity two-version two-phase locking; it allows many readers and one writer to access the same item simultaneously.

- Log records contain only after-images of updates, thus making logs small. Recovery from system failures requires one scan over the log resulting in fast restarts.

In addition to conventional transactions, users can run "hypothetical" transactions. Hypothetical transaction allow users to pose "what- if" scenarios (as often done with spreadsheets). Users can change data and see the impact of these changes without changing the database. Users can also create versions of objects. Ode will track the relationship between versions and provides facilities for accessing the different versions.

Ode 2.0 is described as being used as the multi-media database engine for AT&T's Interactive TV project (however, the current release is 3.0). Ode 1.1 (an older version of

Ode with limited capabilities) has also been distributed to 30+ sites within AT&T and 135+ universities.

3.9.2 ODBMSs Supporting Ada

Commercial ODBMSs have generally been developed to provide interfaces to programming languages such as C++, Smalltalk, and Lisp. Moreover, one of these languages generally plays the role of the ODBMSs "DML", in the sense that the preferred language for writing object methods is one of these languages (in some cases, more than one language can be used). This can present a problem for Department of Defense applications that must (or would prefer to) use Ada for software development. There are two basic approaches to addressing this problem:

- define objects in one of the "native" languages of the ODBMS, and call the methods of these objects from within Ada programs in order to invoke object behavior.

- develop an ODBMS having Ada as its (or one of its) native languages.

The commercial ITASCA Ada interface was described in Section 3.5. However, ITASCA's native language is Lisp. [MR93] describes two ODBMSs having Ada is the native language: Classic-Ada with Persistence, and Science Applications International Corporation's (SAIC) ODBMS developed for the U.S. Air Force's Dental Data System.

Classic-Ada is a product of Software Productivity Solutions Incorporated (SPS). The basic Classic-Ada package is a preprocessor that extends the Ada language to allow class constructs. Objects may be dynamically created, and a message passing mechanism alleviates problems associated with late binding in Ada. Classic-Ada with Persistence is an extension that allows classes to be defined as persistent. Objects created from persistent classes maintain their state between application executions. A fairly conventional object model is implemented. A class may contain instance variables and instance methods. Single inheritance is supported, and classes may override their parent class' methods. Objects reference each other using 32-bit object identifiers. An identifier remains valid across application executions as long as the target object is persistent. The id is not valid across executions for transient objects.

A schema in Classic-Ada is essentially the class structure as defined by the program. Once this structure has been fixed, it cannot be modified without losing all persistent data. Classic-Ada does not support concurrent access, so only one application can open a database at any given time. Because of this "no-server" implementation, all methods are executed by the application. Once a database is opened, there is no difference between persistent and transient objects, and the programmer need not explicitly activate or deactivate persistent objects. However, when a user-defined class is declared to be persistent, all instances of that class must be persistent.

SAIC's ODBMS is a set of Ada packages which defines an ODBMS that allows class hierarchies, dynamic object creation, persistence, and dynamic creation of object methods. This ODBMS is owned by the government and distributable within its agencies. In this system, classes are defined by using a schema file which defines class inheritance and methods. In addition, each class must have its methods (written in Ada) placed into a package and be linked into the database server. Classes and class methods cannot be created dynamically. However, applications can create application-defined complex types dynamically. A complex type is an Ada data structure which can contain pointers to class instances and method objects. Different applications can use the same database as long as they share a common schema. The class schema is built into the database when it is initialized, so all applications have it available.

The SAIC system consists of one server and multiple clients. Each client communicates with the server by passing a shared memory segment that contains an object message. The server accesses the segment and attempts to resolve the message by following the class hierarchy, in a manner similar to that used in Smalltalk. Methods are executed only in the server. Class methods are actually compiled as subprograms in the server. The server allows only one transaction to execute at a time, maintaining serializability at the cost of performance. This limitation also disallows long transactions, and the documentation encourages the use of short transactions for good multi-user performance [MR93]. [Hal93] references a wargame application running on the SAIC object-oriented DBMS. However, disappointing performance was reported. [MR93] also observes that the system does not seem well-suited to large applications.

[Cho93] describes a prototype interface between Ada and the ObjectStore ODBMS. This was based on prototype design work by Object Design, Inc. in 1992. The prototype provides functions and procedures for manipulating databases, transaction management, and persistent declarations ([Cho93] added access to the collection facility to the facilities described in the original design study). The Ada/ObjectStore interface uses Ada pragma[1] interface statements to call C library routines from within Ada applications, and provides access to ObjectStore without significant degradation in performance.

[Moy93] describes work on a portable interface between Ada and several ODBMSs. Specifically, the report describes the problems in developing a common interface for the ObjectStore and ITASCA ODBMSs. The report notes the difficulty in defining a portable interface due to the lack of standards for commercial ODBMSs, and the desire for interface completeness (i.e., the ability to access all the functionality of the underlying ODBMS). The report notes that Ada 9X, which provides object-oriented enhancements on the programming language side, and SQL3, which defines a standard object interface on the ODBMS side, would simplify the problem of providing a portable, complete, Ada/ODBMS interface. (By inference, the ODMG standard would also assist in providing such an interface).

3.9.3 SHORE

[CDF+94, DNSV94] report on the SHORE Persistent Object Store project, which is investigating the exploitation of parallelism to improve the performance of OODBMS applications, corresponding to the use of parallelism in commercial relational DBMS implementations. SHORE does not attempt to solve the problem of automatically parallelizing arbitrary method code (written, e.g., in C++) in an OODBMS. Instead, the goal is to provide system primitives that make it easy for a programmer to explicitly parallelize an OODBMS application. One of the primitives provided is the ParSet (parallel set). [DNSV94] describes how ParSets can be used to parallelize the 007 OODBMS benchmark traversals.

The ParSet facility is a variation of an idea that has appeared in many places before: essentially, it allows a program to invoke a method on every object in a set in parallel. The ParSets of [DNSV94] were inspired by the "Parallel Set" construct proposed by Kilian

---

[1] A *pragma* is a compiler directive.

[Kil92]. According to [DNSV94], this Parallel Set facility is currently being implemented by Kendall Square Research, under the name "ParaSet", on top of the MATISSE OODBMS.

A ParSet is simply a set of objects of the same type (or an appropriate subtype). The Shore Data Language (SDL), which is an interpretation of the ODMG standard data definition language ODL [Cat94a] (see Section 5.1), is used as the type language for ParSet objects. As envisioned by Kilian, ParSets support five basic operations: `Add`, `Remove`, `Apply`, `Select`, and `Reduce`. `Add` adds an object to a ParSet. `Remove` removes an object from the ParSet. `Apply` invokes the specified method on the objects belonging to the ParSet in parallel. `Select` returns the OIDS of ParSet objects that satisfy a specified predicate (evaluating the predicate in parallel). `Reduce` invokes functions on ParSet objects in parallel and through binary operations reduces the results to a single value (computing a scalar aggregate such as `max` or `sum` is an example of a `Reduce` operation).

3.9.4 EXODUS

EXODUS is a database toolkit developed at the University of Wisconsin [CDG+90]. EXODUS components include the EXODUS Storage Manager and the GNU E programming language.

The EXODUS Storage Manager [CDRS89] is a client-server object storage system which provides "storage objects" for storing data, versions of objects, "files" for grouping related storage objects, and indexes for supporting efficient object access. A storage object is an uninterpreted container of bytes which can range in size from a few bytes to hundreds of megabytes. The Storage Manager provides routines to read, overwrite, and efficiently grow and shrink objects. B+tree and linear-hashing based indexes are provided as access methods. In addition, the Storage Manager provides transactions, lock-based concurrency control, and log-based recovery. Applications can access multiple servers in a single transaction. Distributed commits are performed across servers and clients have access to an interface allowing participation in distributed commits managed by an external agent. The EXODUS Storage Manager is used as a component in the TI Open OODB Toolkit, described in Section 3.9.7.

GNU E is a persistent, object oriented programming language developed as part of the EXODUS project [RC89]. GNU E extends C++ with the notion of persistent data, program level data objects that can be transparently used across multiple executions of a program, or multiple programs, without explicit input and output operations.

Other work related to EXODUS included development of a rule-based query optimizer generator [GD87]. EXODUS is no longer being developed, with effort now being concentrated on the SHORE project described above.

3.9.5 Kala

Kala is a Persistent Data Server from Penobscot Development Corporation [OFAQ93]. Kala is not a DBMS. Instead, Kala provides low level object storage facilities. Kala can be used as a component in the development of higher-level components, such as DBMSs, with domain specific functionality being built using Kala's set of primitives. Kala can be used for such applications as the kernel of DBMS products, a substrate for extended file systems, implementation of language persistence, data manager for groupware applications as well as applications which deal with large, complex, and changing volumes of data (text databases, financial distributed transaction systems). Kala is available on Sun platforms (SunOS / 68000 and SPARC), as well as on 80x86/MS-DOS platforms (both Microsoft and Borland compilers and runtimes are supported).

Kala manages an untyped persistent store, implementing the semantics of robust, distributed, secure, changing, and shareable persistent data. Layers built upon the Kala platform can implement the semantics of objects with the same properties. Kala's managed data elements are made out of uninterpreted bits and references. Data elements (called *monads*) are universally uniquely identified. Bits are stored with no overhead. References, represented in memory as native machine pointers, are stored compactly, introducing an average of 2.5 bytes overhead. Kala manages any data that can be represented in machine memory out of these bits and references. Examples include records, dynamically linked graphs and lists, executable code, and encapsulated objects.

Kala can handle data as small as one bit, and as large as the virtual memory (or larger), while being unaware of the data's semantics. It stores and retrieves data over a distributed and dynamically reconfigurable set of Stores. On retrieval, Kala dynamically relocates embedded references to retain the original topological structure of the data, thus preserving referential integrity. Kala also supports active data, physical store management, and automatic archiving.

Kala's execution architecture is that of multiple (communicating) servers and multiple clients. Kala can also be configured in a standalone (single process) mode.

Kala is a fully recoverable system, short of media damage. Recovery from hardware failures can be supported by the layer beneath Kala. Kala primitives can be used to support arbitrary transaction models, including classic short transactions, long (persistent) transactions, nested transactions, shared transactions, pessimistic and optimistic policies, etc. Concurrency control is achieved through two locking mechanisms (short-term and long-term (persistent, shared) locking), with support for atomicity of operations and two-phase commit.

Kala primitives also support arbitrary versioning models, allowing versions to co-exist in split/rejoined networks, various version organization strategies (single-thread, tree, DAG, etc.). Kala primitives provide mechanisms for arbitrary access and update triggers, such as notifications, security checks upon access/update, etc. The Kala primitives can also be used to support a wide range of access control, security and protection models, including revocable access rights, arbitrary access validation routines, etc.

3.9.6 PERCIO

PERCIO [KT91, TKN92] (formerly called *Odin*) is an ODBMS developed at NEC C&C Research Laboratories, Japan. It provides a persistent programming language PERCIO/C++ that provides largely transparent access to persistent objects. The programming language also provides transaction support, collection objects, and query extensions (integrated with C++, much along the lines of the query extensions provided by ObjectStore). Application programs are defined in C++ in a manner somewhat similar to that used in ONTOS DB, and other C++-based ODBMSs: a translator takes C++ (in this case, PERCIO/C++) source code, extracts the class definitions, and registers them in the database. The program is then compiled, and linked with the PERCIO kernel. PERCIO also provides an interactive browser, which can be used to define and display both classes and instances.

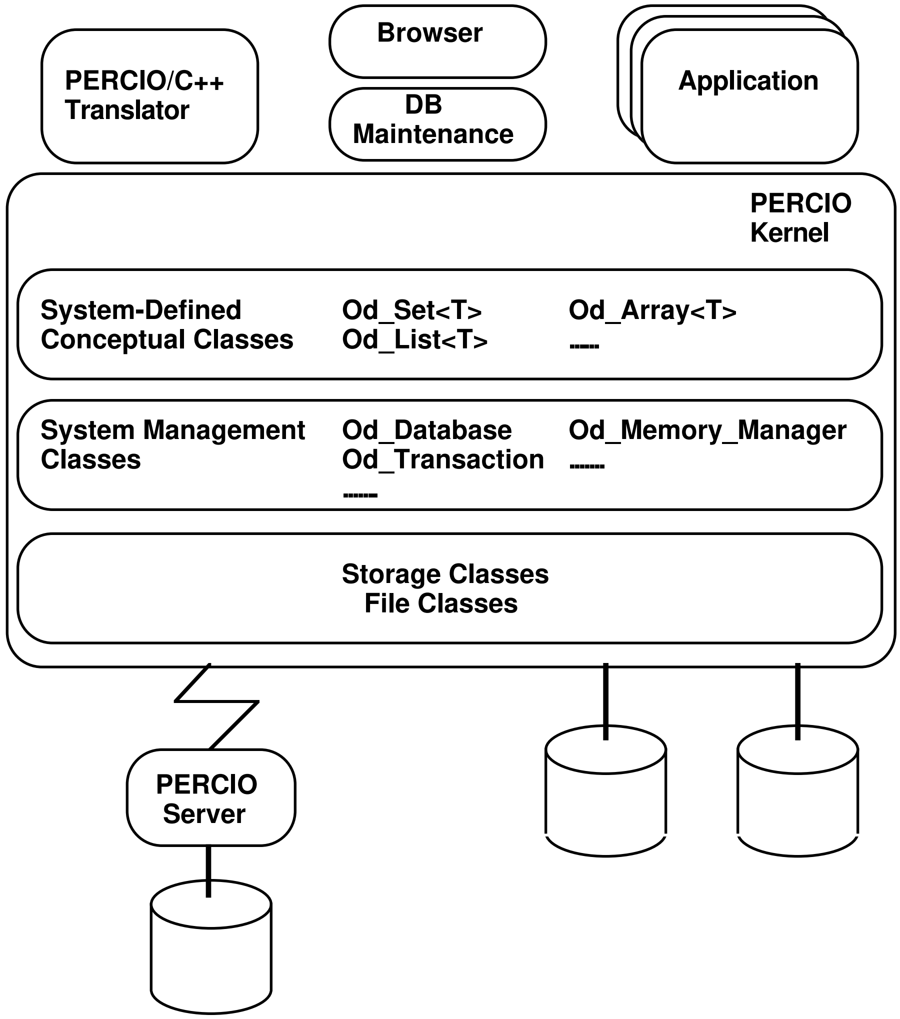


Figure 3.9.6 PERCIO System Architecture

PERCIO itself is defined in terms of layers of object classes, as shown in Figure 3.9.6 [TKN92]. The lowest layer include storage and file classes, which manipulate pages and files in implementing physical object representations. The middle layer includes system management classes that handle memory, transaction, class, and method management. The highest level includes system-defined parameterized classes made available to users, together with the API. These parameterized classes, such as Od_Set<T> (which defines a set object holding instances of class T), provide much of the basic functionality of the DBMS. For example, Od_Set provides methods for adding and deleting members of the set, for getting the first, last, next, or previous member, for selecting members specifying certain predicates, and for joining the set with another set (the latter operations are the basis of the query facilities).

PERCIO provides support for versioned objects, and is unusual in also providing support for view (virtual) object classes, similar to the view facilities provided in relational DBMSs. View objects may be derived from several existing (base) objects, or may be restricted to those objects satisfying a selection predicate.

PERCIO may not become a commercial product, but is reported as being used within NEC for operational network management applications [Tsu93].

3.9.7 Open OODB

Open OODB [TI93, WBT92] is a project of Texas Instruments, sponsored by the Advanced Research Projects Agency, and managed by the U.S. Army Research Laboratory. The objective of the project is to build a high-performance, multi-user object-oriented database system in which database functionality can be tailored by application developers to meet application-specific requirements. The system provides an incrementally-improvable framework that can also serve as a common testbed for research by database, framework, environment, and system developers who want to experiment with different system architectures or components. The Open OODB Toolkit is organized as a collection of independent object services, similar to the approach used in the Object Management Group (OMG) Object Services Architecture [OMG92a,b]. The services can be combined to provide object-oriented database, relational database, repository, and distribution systems. Several modules are database independent, and can be used in non-database-applications.

Open OODB attempts to seamlessly add functionality such as persistence, transactions, distribution, parallelism, or replication to developers' existing programming environments. Open OODB does not require changes to either type (class) definitions or the way in which objects are manipulated. Instead, applications declare normal programming language objects to possess certain addition properties. Such objects then transparently "behave properly" according to the declared extensions when manipulated in the normal manner. For example, if an object is declared to be persistent, the DBMS is responsible for moving it between computational and long-term memory as needed to ensure both its residency during computation and its persistence at program termination. Open OODB extends existing languages (currently C++ and Common Lisp) rather than inventing new database languages.

When an object is declared to the Open OODB to have extended behavior such as persistence, there are certain *invariants* associated with the extension that must be enforced. For example, a remote operation extension might define the invariant that the operator and operands must be materialized in the same physical address space. Invariants can be satisfied in more than one way (e.g., the operations can move to the operands, or vice versa). Each way of meeting an invariant is called a *policy*. When an operation involving an extended object occurs, an Open OODB mechanism called a *sentry* interrupts the operation and transfers control to an Open OODB module called a *policy manager* responsible for ensuring that operations against extended objects "behave properly", by calling appropriate *policy performers* to satisfy the invariants associated with that operation. Each semantic extension is implemented by a different policy manager. Thus, there can be a policy manager for persistence, one for index maintenance, etc. For each type of extension, there are potentially many semantic models (e.g., linear or branching versions), and within each of these, many ways to maintain the invariant (e.g., remote vs. local computation, eager vs. lazy index maintenance). Policy managers can be added independently, and are descended from a common root class to make them type-compatible for invocation purposes. This approach allows new extensions (e.g., time) to be added, the semantics of a given extension to be changed (e.g., nested rather than flat transactions), and implementations of a given policy to be changed or selected dynamically (e.g., object fetch vs. RPC). It also allows the semantic extensions to be hidden from applications. Sentries work by detecting key events such as object dereferences and method invocations and invoking extended behavior, i.e., using concepts of *reflection* or *metalevel computation* (a review of these concepts, together with numerous citations, can be found in [Man93]).

The Open OODB Object Services Toolkit architecture (Figure 3.9.7 [TI93]) is similar to the Object Management Group (OMG) Object Services Architecture (OSA) [OMG92a,b]. Using OSA terminology, Open OODB provides the following services in the current release (0.2):

- event notification service (sentries)
- lifecycle service
- name service
- externalization service
- persistence service
- transaction service
- data dictionary
- query service

The current release runs on Sun workstations, and provides both a persistent C++ with object query language extensions OQL[C++] and a runtime data dictionary, and a persistent Common Lisp. Both languages are provided with recoverable concurrency-controlled transactions based on the Exodus Storage Manager described earlier.

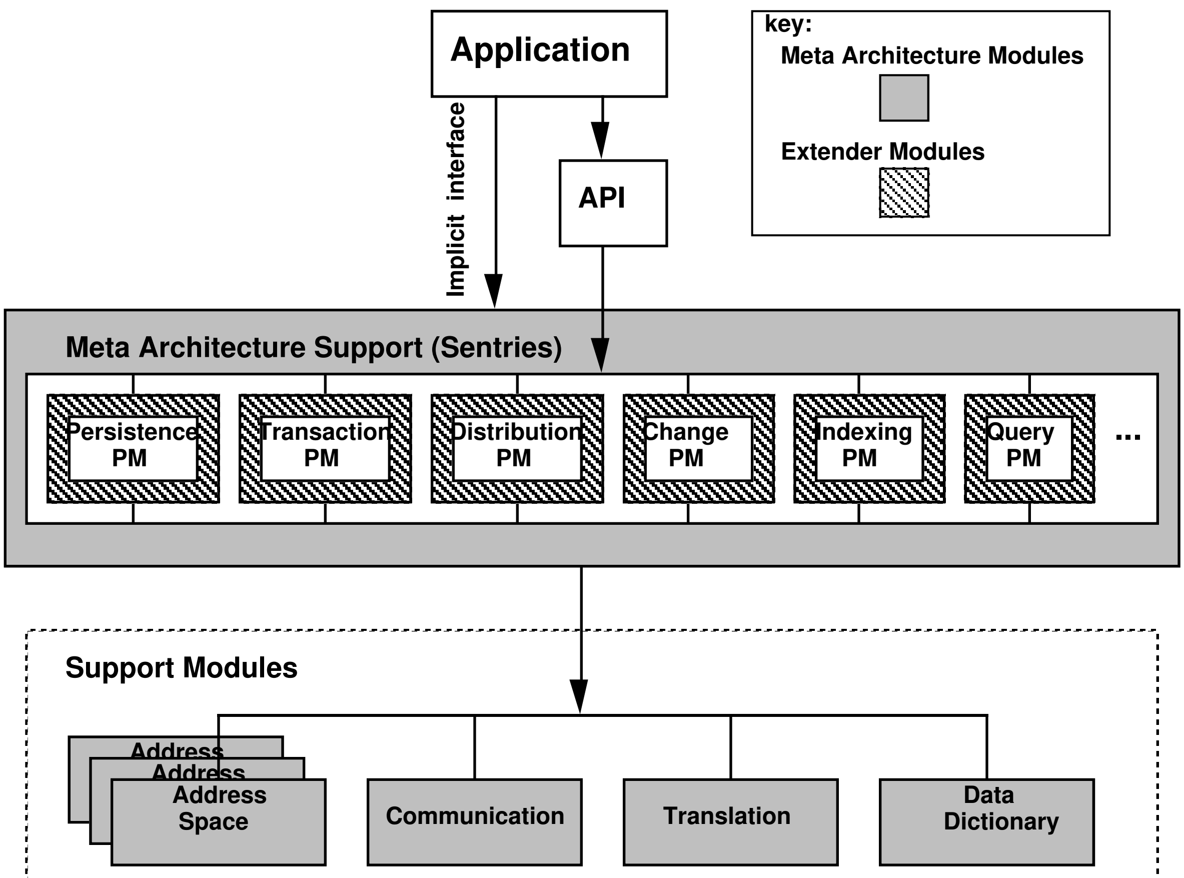


Figure 3.9.7 TI Open OODB Architecture

Release 0.2 includes the following instances of the four basic types of modules:

- Application Programmer Interfaces (APIs) that integrate the functionality of the other modules into a consistent package

    - API [C++] integrates the capabilities of lower-level modules to provide a seamless interface to an extension of C++ supporting persistence, transactions, object query facilities, and a runtime data dictionary. A language preprocessor adds Open OODB functionality to C and C++ source code.

    - API [Lisp] performs similar functions for Common Lisp. Programs that compile under CMU Common Lisp can be extended to use Open ODB features with few code modifications.

- Policy Managers, which provide database functionality by extending the behavior of objects.

    - Persistence PM [C++]

    - Persistence PM [Lisp}

    - Transaction PM [single level; pessimistic concurrency; C++ and Lisp]

    - Transaction PM [single level; no concurrency; non-recoverable; C++] (used for testing)

    - Object Query

- Meta Architecture Modules, which provide definitions and mechanisms (mainly sentries) that enable policy managers to intercede as necessary.

    - Meta Architecture Support [C++] implements sentries for C++.

    - Meta Architecture Support [Lisp] implements sentries for Common Lisp.

    - Language Preprocessor [C++] supports Open ODB extensions for C++.

- Support Modules, providing low-level services.

    - Address Space Manager [non-computational; recoverable; C++ and Lisp] provides recoverable, concurrent access to both C++ and Lisp persistent objects. Storage and recovery is provided by the Exodus Storage Manager.

    - Address Space Manager [computational; C++] manages C++ API objects when they are local to the application.

    - Address Space Manager [computational; Lisp] manages Lisp API objects when they are local to the application.

    - Object Translation [C++] translates between computational and external representations of all C and C++ data types.

    - Object Translation [Lisp] translates between computational and external representations of all Common Lisp data types.

- Data Dictionary [C++ and Lisp] provides internal directory services, naming for C++ and Lisp objects, and run-time type information for C++ applications and Open OODB translation routines (Common Lisp provides its own run-time type information).

The release also comes with the following components:

- Exodus Storage Manager

- CMU Common Lisp (a version modified by Texas Instruments for Open OODB)

There are no explicit communications and distribution modules in Release 0.2. In addition, there are no facilities for indexing, change management, or replication.

4. Object/Relational DBMSs

Object/Relational DBMSs (ORDBMSs) are a new class of DBMSs that, like the OODBMSs described in Section 3, are intended to address the requirements of the kinds of advanced database applications described in Sections 1 and 2. The primary approach taken by ORDBMS developers is that of extending relational DBMS technology with object capabilities in various ways, while preserving full relational capabilities. The products described in this section illustrate a number of different ways of doing this, including both building an object layer on top of a conventional relational DBMS (OpenODB / Odapter), and building a new DBMS from the ground up (Illustra and UniSQL/X). The products also illustrate different system architectures, including both a relational-like query server architecture (OpenODB and Illustra), and a client/server architecture resembling those of several of the OODBMSs described in Section 3 (UniSQL/X). These architectural variants are discussed more fully in Section 6.

The same caveats that applied to the descriptions of OODBMSs in Section 3 apply to the descriptions of ORDBMSs in this section. As before, the goal here is not to be exhaustive in describing these systems, but to merely to indicate their general characteristics, and to show that these ORDBMSs represent a direction for extended DBMS development in competition with OODBMSs (although, as noted in Section 2.2.11, the distinction between these two classes of DBMSs is not easy to characterize).

4.1 OpenODB / Odapter

OpenODB [HP93] is an ORDBMS from Hewlett-Packard. It is intended to provide database capabilities for a wide variety of applications, including both specialized and commercial applications. OpenODB is based on Hewlett-Packard Laboratories' Iris prototype ODBMS [LK86, LV87, LW89, FBCC+87, FACC+89], which was described in [Man89a]. OpenODB stores code as well as data, simplifying applications, and permitting code as well as data to be shared between multiple users and applications. OpenODB uses a client/server architecture, enabling efficient use of available computing power. OpenODB's clients can use a programmatic interface (API) to access information. In addition, OpenODB allows access to existing data and applications using an object-oriented variant of SQL (OSQL). The OpenODB server and all clients are available on HP-UX for the HP9000 Series 700/800 systems and for MPE XL 4.0 or later versions for the HP3000 Series 900 systems. Client software will also be supported using X terminals.

OpenODB is an example of the use of relational DBMS techniques in supporting an object-oriented system, hence its inclusion in Section 4 as an ORDBMS. The OpenODB object model is implemented by an Object Manager, which is based on an extended relational algebra as its computation model. Every schema is mapped to a relational schema with appropriate constraints, and instances of object model constructs are implemented as corresponding relational constructs. OSQL queries are translated into relational queries, and updates become relational transactions. As a result, techniques for tuning relational database systems can be applied in the OpenODB context. [LV87] describes the details of how such an object model can be translated to corresponding relational structures. OpenODB uses a relational database as its Storage Manager for internally stored data and code. The relational database performs the physical file management and database functions such as multiuser concurrency, transaction management, and recovery. OpenODB uses HP's own ALLBASE/SQL relational DBMS for this purpose.

Odapter is a recently-announced further development of OpenODB which essentially decouples the OpenODB Object Manager and other components from the ALLBASE/SQL Storage Manager, allowing the product to be used with other relational DBMSs as Storage Managers. The first release of Odapter allows Oracle 7 to be used as the Storage Manager in place of ALLBASE/SQL. HP has announced its intention to provide Odapter on top of all major third party database products (HP Press Release, 7/18/94), and has also announced that OpenODB will henceforth be named Odapter. Sections 4.1.1 and 4.1.2 describe the facilities of OpenODB; with the exception of references to ALLBASE/SQL, this material also applies to Odapter. Odapter per se is discussed further in Section 4.1.3.

4.1.1 Object Model

The OpenODB object model is based on the *object*, *type*, and *function* constructs. In OpenODB, an *object* is a combination of data and stored code that can operate on the data. An object denotes some individual entity or concept from the application domain being modeled. The basic characteristic of an object that must be preserved in the model is its distinct identity. An object is assigned a system-wide unique object identifier (oid). All user defined types are subtypes of type `UserSurrogate`.

Objects with similar behavior are grouped into collections called *types* (e.g., `Part`, `Person`). Properties of objects, relationships between objects, and operations on objects (behavioral aspects) are all uniformly represented by *functions* (which correspond to *methods* in other object-oriented models). In order to access properties of an object or other objects related to an object, or to perform operations on an object, a function must be evaluated having the object as an argument. The type of an object serves to define what functions may be applied with the object as an argument. A function takes a single object as an argument, and returns a single object as a result. However, the argument and result may be aggregate objects.

Function names may be *overloaded*, that is, functions defined for different types may have identical names even though their definitions may differ. When an overloaded function is applied to a given object, a single specific function must be selected at the time of application. This function is determined by the type of the object to which it is applied. If the object belongs to several types that all have specific functions of the same name, system- or user-defined rules are used to determine the appropriate function to evaluate. These concepts also apply to functions having several arguments. For this reason, applying a function to an object in OpenODB corresponds to sending the object a message in other object models.

The examples below illustrate some of the aspects of OpenODB functions:

- The function `Part_Number(Part p) -> CHAR(20)` allows access to the value of the part number attribute of a `Part` object. Similarly, the function `Advisor(Student s) -> Instructor` allows access to the advisor attribute of a student (whose value is another object).

- The function `Color(Integer x, Integer y, Photo p) -> Color_Value` allows access to the color values at particular points in a photograph. Due to the restriction that functions must only have single objects as arguments, this function is actually shorthand notation for `Color(TUPLETYPE(Integer x, Integer y, Photo p)) -> Color_Value`.

- The function `Location(TUPLETYPE(Connection c, Layout l)) -> TUPLETYPE(Integer x, Integer y)` allows access to the value of the coordinates of a connection in a diagram (note that a function can return a complex result).

- the function `Adjacent(Part p, Part q, Assembly a) -> BOOLEAN` defines a predicate that is true if two parts are adjacent within a given assembly.

Specializations (subtypes) of object types may be defined, forming an inheritance (isa) hierarchy. For example, the declarations:

```
CREATE TYPE Part FUNCTIONS(
    Part_Number CHAR(20)
    Weight FLOAT);

CREATE TYPE 3Dsolid SUBTYPE OF Part FUNCTIONS(
    Union(3Dsolid, 3Dsolid) -> 3Dsolid);
```

define a `Part` object type having two functions, `Part_Number` and `Weight`, and a subtype `3Dsolid` having an additional function `Union` (these declarations are similar to those that would be required for the example in Figure 2.1 of Section 2). Because `3Dsolid` is a subtype of `Part`, any `3Dsolid` object is also an object of the `Part` supertype, and automatically "inherits" the `Part_Number` and `Weight` functions. All user-defined object types are subtypes of the generic type `Object`, and inherit certain operations that can be applied to any object. Since functions are used in OpenODB to represent operations on objects as well as properties and relationships of objects, inheritance hierarchies involve inheritance of operations as well as properties and relationships. Multiple inheritance is supported. For each new type, OpenODB creates an additional pair of type functions, a *predicate* function and an *extension* function. The predicate function returns TRUE if a given object is of the specified type. The extension function returns a bag (multiset) containing all instances of the specified type.

Given the above declarations, a simple query in the OpenODB OSQL might be:

```
SELECT Part_Number(p)
FOR EACH Part p
WHERE Weight(p) > 350;
```

Users may add new data types and operations by defining them as new object types and functions.

Functions in OpenODB may be implemented in several ways. In a *stored function*, output values corresponding to given input values of function parameters are determined by a conventional database search of stored data (such as the `Part_Number` function above). The extension (the stored data) is a database relation containing a tuple (row) for each combination of related input and output values (or object identifiers). In an *OSQL-based function*, the definition of a function is specified using OSQL statements. In an *external function*, the function is implemented in some general-purpose programming language, and linked to the query processor so that it can be called when function evaluation is required. Such functions would typically be used to implement operations on complex multimedia data types, such as a `compress` operation defined for a type `Image`, or the `Union` function defined above. The choice of function implementation is largely hidden from users of the functions.

External functions can be used to access distributed data and code stored outside of OpenODB. This allows development of new applications that integrate existing data and applications. In this approach, the external functions would encapsulate access to the external data sources or code. OpenODB applications could then call these functions via OSQL statements in the same way as functions defined within OpenODB.

Stored functions may be updated, i.e., the mappings from argument values to results can be explicitly specified. This allows, for example, the assignment of values to the properties of objects. The semantics of such updates (and retrievals) are implicitly defined in terms of relational operations on the stored tables. A formal treatment of the mapping of such functions to relational tables is given in [LV87].

4.1.2 Architectural Features

Figure 4.1.1 shows the OpenODB client/server architecture [HP93]. Clients communicate with the OpenODB server over a network. The interface between the clients and the server is transparent to users. The clients and server can also reside on the same machine. An OpenODB client has the following components:

- Interactive Object-Oriented SQL (IOSQL) Interface: This interface allows interactive use of all Object SQL (OSQL) statements, facilitating rapid prototyping and testing. IOSQL provides basic query, administration and editing capabilities.

- Object Browser: The Object Browser provides a graphical interface for exploration of a database's schema and database contents. This tool is designed to increase the speed of application development by making it easier to find reusable code stored in OpenODB. It supports dynamic updates to schema (types) and data.

- Object Application Call Interface (OACI): This interface allows developers to write OpenODB applications using any programming language that can be linked with C (C++, COBOL, FORTRAN, Pascal). The programmatic interface uses OSQL statements passed as parameters and does not require preprocessors.

- User Applications and Tools: These are OpenODB clients developed using IOSQL, the Object Browser and the OACI.

The OpenODB server consists of the following components:

- Object Manager: This is the central server component. The Object Manager is the query and update processor of the DBMS, and implements the OpenODB object model as described above. The Object Manager executes OSQL calls made by OpenODB clients. The Object Manager processes requests and accesses data and code from the internal data storage manager (Relational Storage Manager), or passes the request to a subsystem outside of OpenODB (External Functions).

- Storage Manager: OpenODB uses HP's ALLBASE/SQL commercial relational DBMS as its Storage Manager for internally stored data and code. The relational database performs the physical file management and database functions such as multiuser concurrency, transaction management, and recovery. Relational database tools are available to help perform online backup and recovery, manage physical distribution of files, maximize availability, and change database parameters.

- External Functions: External functions allow access to data and code stored outside of OpenODB, regardless of data format or location. They are implemented as subroutines written in general-purpose programming languages and compiled outside of OpenODB. With external functions, existing applications can be encapsulated. External functions can be called by any OSQL statement, allowing use of remote data and application code in the same way as any other object.

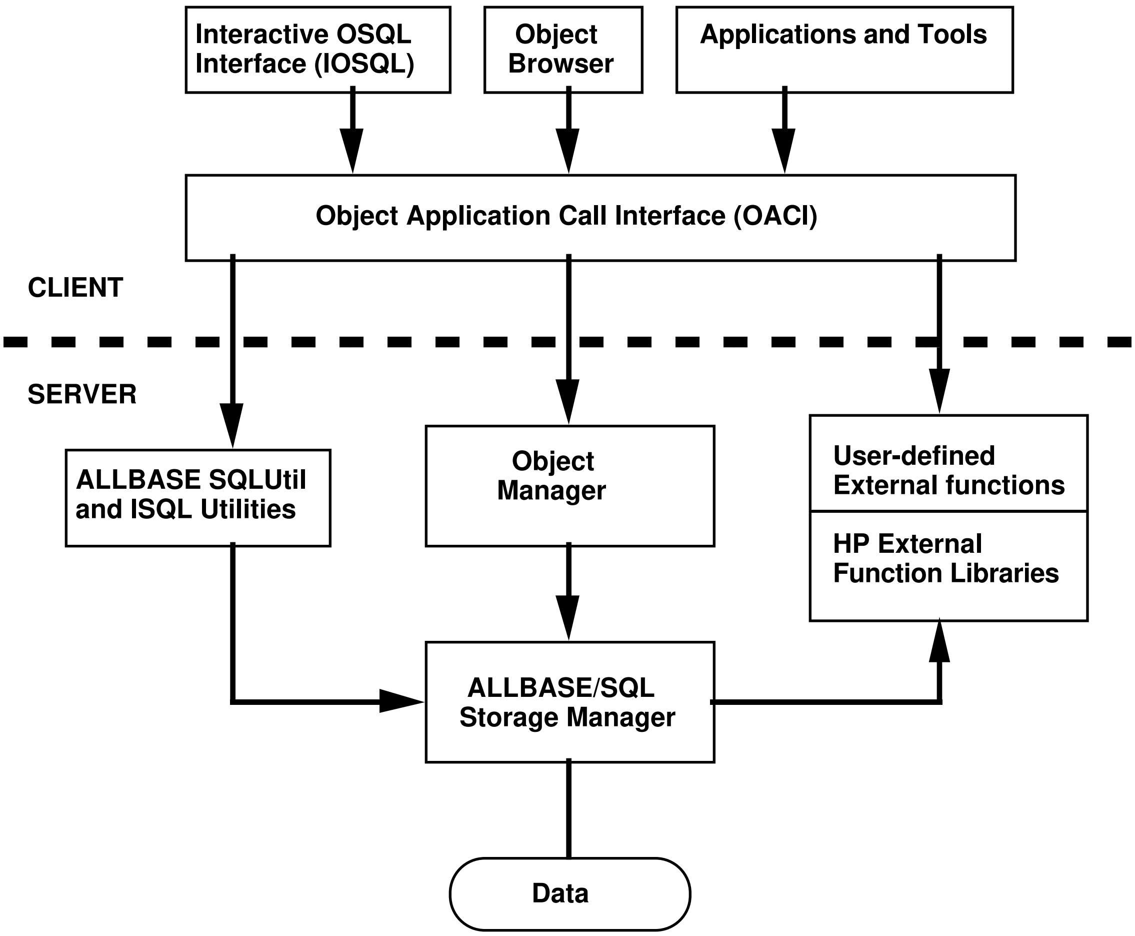


Figure 4.1.1 OpenODB System Architecture

OpenODB provides conventional transaction management facilities, including savepoints. In addition, OpenODB allows specification of one of three isolation levels for each transaction:

> repeatable read--the system uses locking to prevent concurrent users from changing data being selected or updated by the transaction.
>
> read committed--the system uses locking to ensure that a transaction only retrieves committed data; however, it does not prevent other transactions from updating this data immediately after it is read.
>
> read uncommitted--the transaction is allowed to read data without setting any locks.

OpenODB also has a robust logging and recovery facility [EH92]. In case of a failure, OpenODB can handle rollback or rollforward recovery to a particular time, using a log file to recreate saved work. OpenODB also supports online backup, and can take advantage of system features such as disc mirroring, if available.

Indexes are automatically defined on object identifiers (Oids) when types and functions are created. Users can also define their own indexes. Related functions can also be clustered together (e.g., stored in a common database table) to improve performance.

Access to OpenODB can be controlled at the database and function levels based on individuals or a group of users. Authorization statements provide a flexible way to control access to types and functions in OpenODB.

OpenODB supports storage of large, unformatted data, such as graphics, images, and voice, in binary format. OpenODB functions can then be defined to manipulate this information.

OpenODB also supports dynamic schema modification, allowing creation of new functions and types at runtime. The implementation of functions can also be changed without having to recompile applications. Also, an object can belong to more than one type, which allows the type of an object to dynamically change without having to destroy and recreate the object.

Open ODB provides a C++ interface in the form of a class library. This class library provides:

- session and transaction control facilities, such as connecting to databases, and commiting or rolling back transactions

- facilities to execute OSQL statements from within the C++ application

- facilities to define cursors to scan the output of OSQL statements

- facilities to associate a C++ object with an existing OSQL function, and use the object methods to execute the OSQL function to access/modify the database

- C++ classes that represent the OSQL data types, and can be used to pass values to and receive values from OSQL functions

Users can also define user-defined types in C++. These consist of surrogate objects in C++ (of class `ODBSurrogate`) that contain references to corresponding OpenODB objects (a surrogate object contains no other internal state). The user then constructs C++ objects of class `ODBfunc` that act as surrogates for OpenODB functions, and calls these functions from within C++, passing object surrogates to them, to operate on the database. Thus, the interface is not quite as transparent as those provided in some of the C++-based ODBMSs described in Section 3.

OpenODB also provides an interface to ParcPlace Smalltalk, in the form of a Smalltalk class library. This interface has two levels. The lower level provides a one-to-one mapping between Smalltalk methods and equivalent C-function calls in the OpenODB OACI. The higher level encapsulates those methods into a simpler set of operations that Smalltalk applications can directly call. The primary class at the higher level is class `OpenODBConnection`. Instances of this class are objects representing connections to an OpenODB database, through which OSQL operations can be sent to the database, and results returned to the Smalltalk application. Results can be returned either in the form of a string, or as an object having the form of an OpenODB session variable or Smalltalk `OrderedCollection`. A simple example of the use of this interface is shown below.

```
|  dbObj res |

dbObj := OpenODBConnection new.        //create an OpenODB connection

dbObj
  environment: 'dbe';
  username:  'tom';
  password:  'secret';
  connect.

res := dbObj evaluate:  'select room#(r), roomtype(r)  //execute query
           for each roomprofile r;'.

Transcript show:  res printString; cr.             //print the results

dbObj commit; disconnect.              //end the transaction and disconnect
```

OpenODB also provides an interface, called EDA-Objects, to Information Builders, Inc. Enterprise Data Access/SQL (EDA/SQL) Application Interface product. Using EDA-Objects, users can integrate information from the many different database products (including both relational and hierarchical systems) that can be accessed via EDA/SQL, and manipulate this information as objects in OpenODB. EDA-Objects consists of a set of OpenODB external functions which provide access to EDA/SQL facilities, and return results as OpenODB types.

The primary means of making Open ODB's object model visible to users is the Object SQL (OSQL) interface. The OSQL interface is an attempt to combine the capabilities of the OpenODB model with SQL. The three main areas in which OSQL goes beyond SQL in adapting to a model based on objects and functions are:

- Users manipulate types and functions rather than tables
- Objects may be referenced directly rather than indirectly, through their keys. Interface variables may be bound to objects on creation or retrieval and may then be used to refer to the objects in subsequent statements.
- User-defined functions and OpenODB system functions may appear in `WHERE` and `SELECT` clauses.

OSQL is a complete language with statements that allow users to define and manipulate OpenODB databases, specify authorization by individuals or groups, define transactions, embed program logic within functions, and administer databases.

The following is an example illustrating the use of OSQL in defining and using the following schema [OFAQ93]:

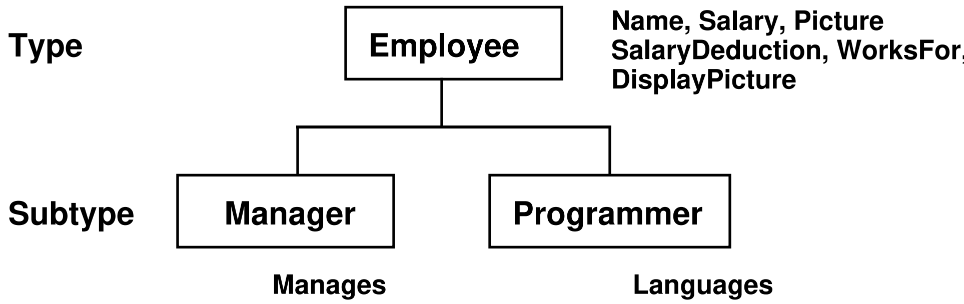


Figure 4.1.2 Example OpenODB Schema Diagram

Create a user-defined type called `Employee` with three stored functions:

```
CREATE TYPE Employee
  FUNCTIONS (Name CHAR,
             Salary FLOAT,
             Picture BINARY);
```

Create type `Programmer`, a subtype of `Employee`. Also define the stored function `Languages` on `Programmer`:

```
CREATE TYPE Programmer SUBTYPE OF Employee
  FUNCTIONS (Languages SETTYPE(CHAR));
```

Create type `Manager`, a subtype of `Employee`. Also define the stored function `Manages` on `Manager`:

```
CREATE TYPE Manager SUBTYPE OF Employee
  FUNCTIONS (Manages SETTYPE(Employee));
```

Create an OSQL-based function `SalaryDeduction` on `Employee`:

```
CREATE FUNCTION SalaryDeduction (Employee e) -> FLOAT AS OSQL
   SELECT (0.3 * Salary (e));
```

Create an OSQL-based function `WorksFor` on `Employee` to define the relationship with `Manager`:

```
CREATE FUNCTION WorksFor (Employee e) -> Manager AS OSQL
   SELECT mgr
    FOR EACH Manager mgr
    WHERE e IN Manages(mgr);
```

Create an external function on `Employee` to display the employee's picture:

```
CREATE FUNCTION DisplayPicture (CHAR Name) -> CHAR AS EXTERNAL
  SIMPLEEXTFUN('DisplayPicture $Name');
```

Put data into three stored functions defined on the `Programmer` type:

```
CREATE OBJECT AS Programmer
 FUNCTIONS (Name, Salary, Languages)
```

```
 :bob ('Bob Cox', 55000, SET('PL/1', 'C')),
 :sue ('Sue Smith', 65000, SET('COBOL'));
```

Put data into three stored functions defined on the `Manager` type:

```
CREATE OBJECT AS Manager
 FUNCTIONS (Name, Salary, Manages)
 :al ('Al Ott', 70000, SET(:bob,:sue)),
 :jim ('Jim Hill', 100000, SET());
```

Put data into three stored functions using a reference to another object:

```
CREATE OBJECT AS Manager
 FUNCTIONS (Name, Salary, Manages)
 :chris ('Chris Jones', 80000, SET(:al,:jim));
```

Having created the database, the following queries can then be specified:

Select Bob's salary:

```
SELECT Salary (:bob);
  Result: 55000
```

Select the names of the employees that Al manages:

```
SELECT Name(Manages(:al));
  Results: 'Bob Cox'
           'Sue Smith'
```

Select the names of all managers:

```
SELECT Name(m)
 FOR EACH Manager m;
    Results: 'Al Ott'
             'Jim Hill'
             'Chris Jones'
```

(The ":" denotes a temporary variable (e.g. `:bob`) defined in the application to store a reference to an OpenODB object.)

OSQL uses the functional notation `Name(m)`, corresponding to the OpenODB functional model, rather than the corresponding dot notation `m.Name` used in some object SQL variants. The meaning is, however, the same. This functional notation extends to the use of *nested functions* in place of the path expressions used in some object query languages, as in `Name(Manages(:al))` above, which has the same meaning as would the path expression `:al.Manages.Name`.

OSQL includes programming flow statements, including IF/THEN/ELSE, FOR and WHILE. An example of an OSQL procedure using these statements is:

Create a Procedure:

Convert all managers to programmers that have less than a specified number of employees:

```
CREATE FUNCTION MgrToEng (INTEGER minemps) -> BOOLEAN
```

```
AS OSQL
   BEGIN
     FOR m IN Manager DO
        IF (COUNT(Manager (m)) < minemps)
        THEN
           BEGIN
             ADD TYPE Programmer TO m;
             REMOVE TYPE Manager FROM m;
           END
        ENDIF;
    END;
```

Executing the Procedure:

```
CALL MgrToEng(3);
```

This procedural capability allows OpenODB functions to be quite complex, allowing large portions of application code to be moved into the database. This facilitates code sharing, and centralized control of key database behavior.

4.1.3 Odapter

As noted earlier, Odapter is a further development of OpenODB which allows OpenODB's object facilities to be used with other relational DBMSs as Storage Managers. The first release of Odapter allows Oracle 7 to be used as the Storage Manager in place of ALLBASE/SQL. In other words, the first Odapter release can be thought of as being exactly the same (and having the same facilities) as OpenODB as described above, but substituting Oracle for ALLBASE/SQL as the Storage Manager, and replacing ALLBASE/SQL utilities with Oracle utilities, in Figure 4.1.1. Figure 4.1.3 shows the Odapter architecture as a revised version of Figure 4.1.1, with the components provided by the particular relational DBMS being used as the Storage Manager highlighted. HP has announced its intention to provide Odapter on top of other major relational database products in addition to Oracle, and has also announced that henceforth OpenODB will be named Odapter.

Using Odapter as a front end, current Oracle databases can be made to look "object-like". If new object classes are defined in OSQL, Odapter converts them to relational tables in Oracle. In addition, like OpenODB, Odapter comes with class libraries for C++ and Smalltalk. This allows development of new object-oriented applications with C++ or Smalltalk, using Odapter as the front-end and Oracle as the back-end. The result is that users can maintain current legacy applications in the same database as new object-oriented applications.

Using ParcPlace Systems' VisualWorks development environment, Odapter can automatically create Smalltalk classes from OSQL object classes and access them from Smalltalk. Odapter is also integrated with HP Distributed Smalltalk, an implementation of the Common Object Request Broker Architecture (CORBA) specifications that support the development of distributed object applications (see Section 5.3).

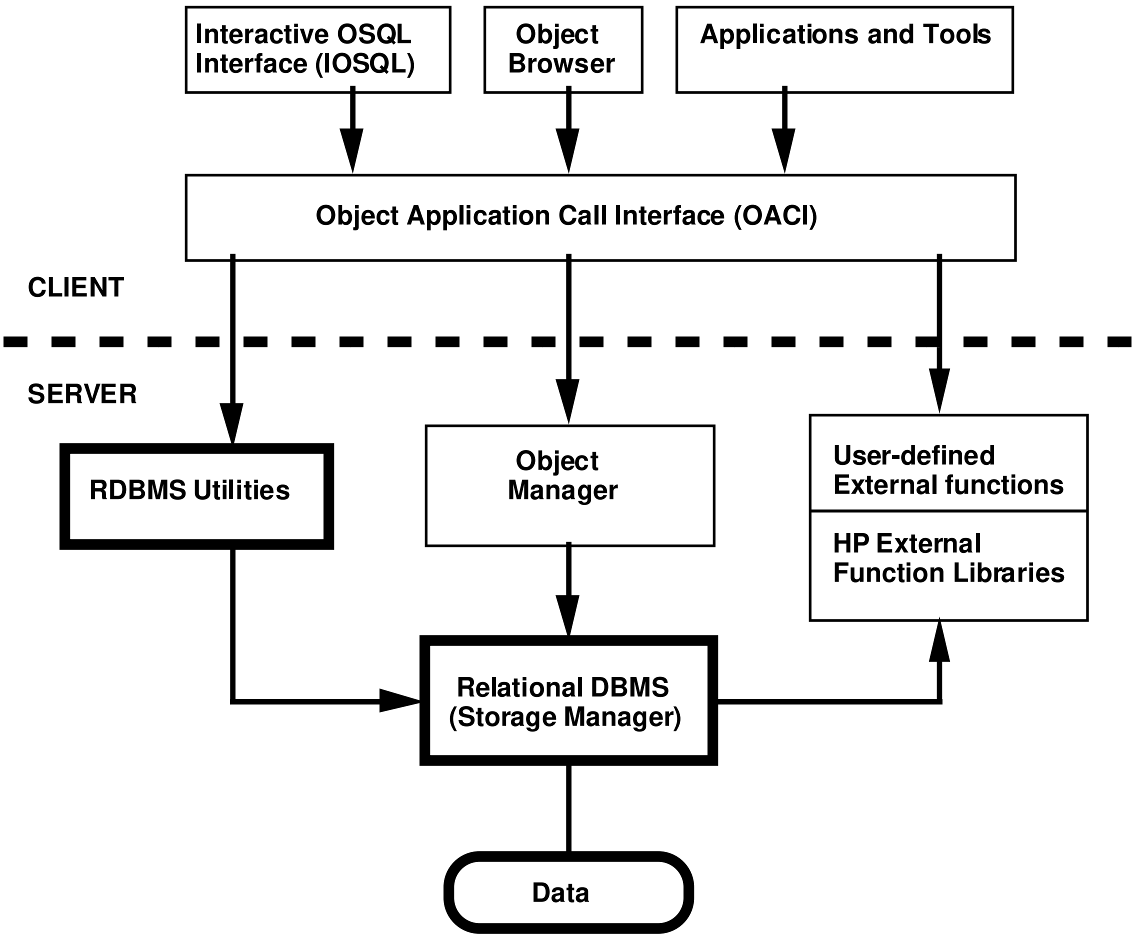


Figure 4.1.3 Odapter Architecture

Odapter client software will be available on HP 9000, Sun Sparcstation and IBM RS/6000 Unix workstations and Windows-based PCs; it will be available in a server configuration on HP 9000 servers and workstations running Oracle release 7.0 or later. In addition, Odapter is compatible with a wide variety of object-oriented software development environments and tools, including HP SoftBench, HP Distributed Smalltalk, ParcPlace Smalltalk and VisualWorks, IBI FOCUS and EDA/SQL, ProtoSoft Paradigm Plus, STEP Tools EXPRESS Tool Kit, Hitachi ObjectIQ, and HP's COMPASS class library.

4.2 Illustra

Illustra [Mon93, Haw94, Ube94] is an object-relational database management system (ORDBMS) available from Illustra Information Technologies, Inc. (Both the company and the product have gone through several name changes, specifically from "Miro" to "Montage", and now "Illustra"). Illustra represents the commercialization of the seven-year POSTGRES research project [SR86, Sto86a].

Illustra supports inheritance, user-defined types, functions, and operators, ad-hoc queries, time travel, a rules system, tertiary storage devices, and very large typed objects, among other things. Illustra supports SQL, together with extensions to provide object-oriented facilities. These extensions are based on the on-going SQL3 standardization activities.

User functions may be written in C or in SQL. New types of access methods and storage managers are also easily added to the system. The query optimizer can use declarations associated with new access methods and user-defined functions to determine when to choose a new access method and the cost of executing a user-defined function [Ube94]. C functions can be dynamically loaded into the database server on demand, and either kind of function may be executed from the query language. Illustra is currently in release 2.0. Current server platforms supported are Silicon Graphics, SunOS/Sparc, Solaris/Sparc, DEC OSF/Alpha, with ports in progress to Windows NT and Solaris/Intel. Clients can run on Unix or Windows systems.

4.2.1 Object Model

Illustra implements the relational standard SQL92, together with object extensions for defining user-defined data types and functions from the ongoing SQL3 standards committee activity. These extensions include type and table hierarchies, multiple inheritance, object identifiers, and function overloading. Because the SQL3 activity is ongoing, and changes to SQL3 are being made continually, the facilities implemented by Illustra are not necessarily totally synchronized with the current state of SQL3. However, Illustra is committed to being compliant with any resulting SQL3 standard. See Section 5.2 for further discussion of SQL3, and these issues.

The basic idea behind the object extensions of the relational model implemented by Illustra is that, in addition to the normal built-in types defined by SQL, user-defined *abstract data types* 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. User-defined types may be either *base types* having no components, or *composite types*, having attributes. Moreover, a table can be thought of as being a collection of instances of a user-defined composite type.

In Illustra, the `create type` statement is used to define a new type. There are several syntactic variants of this statement. To create a new *base type*, the user must provide the name of the new type, a description of the internal length of an object of the type, and the registered input and output functions for the type. For example:

```
create type circle_t
(
      internallength = 24
      input  = circle_in
      output  = circle_out
);
```

The `input` function converts the external (text) representation of the type to an internal representation which can be manipulated by operations defined for the type. The `output` function does the reverse conversion. The `input` and `output` functions must be registered with the system using the `create function` statement before creating the new type. These declarations for type `circle_t` might be:

```
create function circle_in(text)
returns circle_t
as external name '/tmp/circle.so'
language C;

create function circle_out(circle_t)
```

```
returns text
as external name '/tmp/circle.so'
language C;
```

An `under` clause (as in `create type special_circle_t under circle_t)` is used if the type being created inherits behavior from an existing type. This inheritance capability is similar to that ordinarily found in object-oriented systems (multiple inheritance is also supported).

Modifiers can be specified with such type declarations to allow specification of additional support functions and optional information about a user-defined base type. For example, for user-defined large object types, the `import` and `export` modifiers can be used to define functions that read and write these values to and from data files. (Illustra provides its own `import` and `export` functions for the built-in `large_object` and `large_text` types, `FileToLO` and `LOToFile`, respectively). Similar facilities are available for binary large objects. The `passedbyvalue` modifier specifies that the type is passed by value (rather than by object reference). The `cannothash` modifier specifies that type equality cannot be determined by a bitwise comparison.

To create a new *composite type*, the user must specify the names and data types of the new type's components. For example:

```
create type person_t
(
      fname text,
      lname text,
      age int,
      sex char(1)
);

create type employee_t
(
      manager ref(employee_t),
)
under person_t;
```

Type `employee_t`, is defined as subtype of type `person_t`, having an additional `manager` column. Composite types cannot be used recursively. That is, because `employee_t` is a composite type, it may not have a component of type `employee_t`. However, a reference to an object of type `employee_t` (type `ref(employee_t)`) can be a component of an object of type `employee_t`, hence the definition of `manager` in the above example.

A third form allows creation of a new type from an existing type:

```
create distinct type <type_name> as <type_name>
```

This allows creation of a new type that has the same representation as an existing type, but is considered by the system as distinct.

Operators for user-defined types, as well as other functions, are defined using the `create function` statement, which registers a new function in the database. The function body may be written in SQL or C. The general syntax is:

```
create function <function_name> [<parameter_list>]
```

```
returns <type name>
[with <modifiers>]
as <sql_statements> | external name <file_path_spec>
language {C} [[not} variant]
```

The query language parser and query execution procedure use this information in parsing query expressions containing references to new types (and their associated operators), optimizing the queries, and calling the appropriate routines as needed. The following examples of the modifiers that may be specified illustrate some of the information that may be useful in query optimization, and how it can be used:

`negator` = `<function_name>` `<parameter_list>` specifies a function which is the negator of the function being registered.

`hashable` specifies that the function being registered is hashable.

`handlesnulls` specifies that the function should be called even if one or more of its arguments is NULL. If `handlesnulls` is not specified, the default action is for the function not to be called, and a result of NULL to be returned.

`leftsort` = `<function_name>` `<parameter_list>` specifies a function which is used to sort the left side of a sort merge on the function which is being registered. The function must return a Boolean value.

`untrusted` specifies a function that is intended to be executed on the client rather than the server. This applies only to C functions.

`percall_cpu` = `<integer_constant>` specifies the CPU cost per function invocation of the function being registered, compared to the per tuple cost had the function not been invoked. This applies only to C functions.

`selfunc` = `<function_name>` `<parameter_list>` specifies a function which estimates the selectivity the function which is being registered. The function returns a floating-point value between 0 and 1. This applies only to C functions.

`late` specifies late binding for the function being registered. This applies to functions which take one or more arguments of a row type, and are overloaded in an inheritance hierarchy. If late binding is specified, the system applies the best match of an overloaded function when the function is invoked on the supertype of a table. The system examines each row individually and attempts to match the most applicable function on a per row basis.

If the body of the function is written in SQL, the text of the function is contained inside the `create function` statement as one or more SQL statements. If the body of the function is written in C, the function must be compiled and linked to create a dynamically loaded executable. In this case, the `external name` clause gives the pathname of the file containing this executable. A C function is by default assumed to be variant, which means that under certain circumstances it may return different values when invoked with the same arguments (e.g., a function which performs calculations based on the current date or time). If a function is defined as `not variant`, and if it is expensive (the `percall_cpu` or `perbyte_cpu` value is greater than 0), the optimizer may cache the results of the function.

In Illustra, the `create table` statement is used to create a table in the database. Tables can be defined by specifying the table name and its columns, as in a conventional relational system. An example might be:

```
create table journalists
(
      name text,
      company text
);
```

Columns in such tables can be defined as being of user-defined types, as well as base types. The type of the `journalists` table is anonymous. A table can also be based on an existing composite type, as in:

```
create people of type person_t;
```

In this case, the table gets its columns from the definition of `person_t`, as defined above. Finally, a table and a new type can be created in a single statement[1]. For example, both type `person_t` and table `people` could be created by the single statement:

```
create table people of new type person_t
(
      fname text,
      lname text,
      age int,
      sex char(1)
);
```

For each table that is created, Illustra adds eight columns which it uses to manage the table. These columns are:

- oid: contains a 64-bit unique identifier for the row (object); these oids are preserved by the DBMS and not reused so they can be referenced in applications
- tmin: the commit time of an insert or update transaction
- tmax: the commit time of a delete transaction
- xmin: the identity of the inserting transaction
- xmax: the identity of the deleting transaction
- cmin: the command identifier of the inserting transaction
- cmax: the command identifier of the deleting transaction
- ctid: the relative tuple id within the table

[1] The ability to base an entire table directly on a user-defined type, and to bundle the creation of a new table and a new type in a single statement, was part of the SQL3 language when Illustra was being developed, but has since been removed from SQL3. This particular part of SQL3 is in considerable flux. See Section 5.2 for further discussion.

The values for these columns are not returned when a user executes a `select *` on the table, but they can be selected when explicitly named in a query target list, or qualified on when named in a search condition.

Inheritance applies to tables as well as to types. A table can be defined as a subtable of an existing table by specifying an `under` clause in the subtable specifying the supertable. The subtable then gets its structure from the supertable. In defining a subtable, the relationship between the supertable(s) and the subtable must be parallel to the relationship between the type of the subtable and its supertype(s). For example, for the definition

```
create table employees of type employee_t under people;
```

to be legal, type `employee_t` must be defined as a subtype of the type (`person_t`) underlying the `people` table.

In addition, when a subtable is created under a supertable, the data in the subtable is visible when a user queries the supertable. That is, rows (objects) in the subtables are treated as if they are also contained in the supertable. For example, given the `people` table and its subtable `employees` as defined above, the query

```
select * from people;
```

returns the first name, last name, age, and sex of all the objects in the `people` table, together with the first name, last name, age, and sex of all the objects in the `employees` table (and also the values of those same columns for any other subtables created under the `people` or `employees` tables). If the results should be limited to only those in the specific table being referenced (and not any of its subtables), the keyword `only` must be specified, as in

```
select * from only (people);
```

It is also possible to query a table using a correlation variable, so that the results include not only the columns from the supertable, but also the additional columns defined for the various subtables. Such a query returns rows of different lengths. For example

```
select p from people p;
```

returns the "people columns" from the `people` table, and the "people columns" plus the "employee columns" from the `employees` table.

Indexes are defined by the `create index` statement. An index can be built on a single column of a table, or on a function involving one or more columns in the table being indexed. A functional index improves the performance of queries which qualify on the value returned by a function. When a functional index is used, Illustra precomputes the value of the function and stores it in the index. If the data in the table changes so that the value of the index changes, the system automatically updates the index. A functional index can be used to simulate multi-column indices if that features is required. For example:

```
create index exchange_date on exchange
using btree(exch_time);
```

creates a `btree` index named `exchange_date` on the `exch_time` column of the `exchange` table. Similarly:

```
create index zone_func_ind on zones
using btree(Area(dimensions));
```

creates a btree index named `zone_func_ind` on the `zones` table. This is a functional index which is computed by a defined function named `Area(dimensions)`. The function computes the area of a zone from the `dimensions` column of the `zones` table. This index improves the performance of queries in which the search condition is based on the area of a zone.

The `using` clause specifies the access method for the index. The access method for built-in types is `btree`. Additional access methods are available (or can be constructed) for use with data types for which a `btree` is not the most efficient access method. Some of these are mentioned in the next section.

4.2.2 Architectural Features

The Illustra ORDBMS includes the Illustra Server database engine, the Illustra Viewer -- a visualization tool that simplifies queries of complex data, and Illustra DataBlades -- specialized modules that extend the capabilities of the database for specific applications.

The Illustra Server extends the relational database model through its ability to handle complex information, and the inclusion of object-oriented facilities and capabilities. It uses the familiar relational row-column metaphor for all data, so that text, numbers and complex data are all viewed, managed, manipulated and queried the same way. The relational metaphor is extended to allow data of any size and complexity to be stored and accessed in the way that is most effective. SQL, used to access and manage data, is extended with SQL3-based capabilities to allow the definition of user data types and functions.

The Illustra Viewer uses visualization technology to organize information in visual terms -- by location, shape, color and intensity, for example. Similar to a "flight simulator," the Illustra Viewer allows the user to visually navigate through data, refining each step by "panning" and "zooming" with a mouse.

A DataBlade is a combination of data types, functions (methods) defined for those types, and access methods customized for those types and functions, that are designed to support a specific application area. The name comes from the idea that the DBMS can be thought of as a "razor" into which technology-specific "DataBlades" are inserted to support various applications. Text, Spatial, and Image are the first of many DataBlades that will comprise a full-range of industry-specific products created by Illustra, third parties and users based upon their own expertise. Specifically:

- The Image DataBlade supports image conversion, storage, manipulation, enhancement and management of more than 50 image formats, and performs optional automatic conversion of formats.

- Points, lines, polygons and their spatial relationships are supported with the Spatial DataBlade. The DataBlade defines nine basic spatial types and over 200 SQL functions for use on spatial data, and supports the R-Tree access method for high speed navigation of spatial data.

- The Text DataBlade expands the database's functionality by adding new data types and functions that manage text and document libraries, together with a new access method (Doc-Tree) which provides full text search facilities.

For example, the Text DataBlade allows conditions involving full text search to be combined in relational queries with access to other data types, as in [Ube94]:

```
SELECT *
FROM Employees
WHERE Contains(resume, 'sql or quel and programming')
      AND dept = 'Marketing';
```

DataBlades for OCR, time series, and isochronous (real-time) data are under development. The latter DataBlade will include a special storage manager to optimize data delivery.

The API for the Illustra server is called LibMI. It is used for the development of applications which access data stored in a Illustra database from the client or the server. LibMI also provides a direct access interface which bypasses the Illustra query processor, executor, and optimizer. This interface can be used by sophisticated developers who are familiar with LibMI and database access methods.

The basic tasks of a LibMI program are to send SQL command strings to the Illustra server for execution and to process results returned by the Illustra server to the application. These SQL commands can, in turn, invoke user-defined methods stored in the database. Thus, the architecture provides the ability to develop user-defined data types in the database, but without the tight coupling of the database objects to the application programming language found in many of the OODBMSs described in Section 3.

Applications can run on either the client or the server. Both client and server LibMI applications can support up to 32 connections to the same database. In a client application these connections can be to different servers or the same server. In a server application all the connections are to the same server.

A LibMI application can be executed from a server as well as from a client with little or no modification to the application code. Any function which does not require interactive input from the user application can be written as a server function. Since a server application runs as part of a Illustra server process, the application must be careful to not do anything that might negatively affect the running of the server, such as exiting.

Unlike a client application, which can simultaneously connect to several databases, a server application executes in the context of a server process which already has a database open and is inside a transaction.

A C function can be created as an untrusted function by using the `untrusted` modifier with the `create function` statement, as described above. An untrusted function is one which is run on the client rather than the server. Untrusted functions are used whenever the user wants the function to execute in the client process (e.g., because it is being debugged, or is a display function, and is dependent on the client's window manager). When a function is registered as untrusted, it is not linked into the server code. When the function is invoked by the DBMS, it is actually executed by the client, which returns the results to the DBMS.

Illustra provides a number of built-in facilities for handling large objects (BLOBs). Objects that are larger than approximately 8K can be stored as instances of one of Illustra's built-in large object types, `external_file`, `large_object`, or `large_text`. External files are useful for situations in which an object, such as a picture or video, is created and accessed frequently, but is rarely or never updated through Illustra. External files can be accessed quickly, but the cost of this speed is that they have no support for transaction rollback, crash recovery, or multi-user protection (concurrency control). External files can also be accessed by applications other than Illustra. A large object is stored using the Illustra DBMS facilities. It has all the database properties of any Illustra object. Several rows in a database can point to a single `large_object`. Illustra retains reference count information on how many rows refer to the object. An object with zero references to it is automatically deleted when the database is *vacuumed*, unless it is specified as being *archived*, in which case it is moved to archival storage (these Illustra facilities are discussed further in Section 6.5).

When a user selects a `large_object` from a table, the returned value is a *handle* to the large object. A handle is a character string which represents the object. When a user selects an `external_file` from a table, the returned value is the pathname of the file. When a user selects a `large_text` from a table, the returned value is the textual contents of the file.

LibMI contains an interface for manipulating large objects. Examples of the operations defined for large objects include:

`mi_large_object_create()`--Create a `large_object` or `external_file` from a file given a file name, and return a handle to the object (this handle can be used immediately, or stored in a column of a table as a persistent reference to the object).

`mi_large_object_open()`--Opens an external file or large object for processing, given a handle to the object or file. A large object must be open to be accessed.

`mi_large_object_gethandle()`--Given the name of an external file, returns a handle which can be passed to `mi_large_object_open`.

`mi_large_object_info()`--Returns a pointer to a predefined structure which contains information about the large object or external file referenced by the specified handle.

`mi_large_object_seek()`--Set the position for the next read or write operation to or from an external file.

`mi_large_object_read()`--Read data from an external file or large object from the current position.

`mi_large_object_readwithseek()`--Read data from an external file or large object from the specified offset.

`mi_large_object_write()`--Write data to an external file or large object from the current position. Currently, writing to objects of type `large_object` is limited to writes of empty large objects (appending is not currently supported) [Mon93].

Illustra also implements a general rules and alerters system. Rules may be placed on data retrieval as well as on data modification statements. Alerters can be used to signal other processes when a rule has fired [Ube94]. Details of these mechanisms can be found in [Mon93].

4.3 UniSQL/X

UniSQL, Inc. offers a suite of integrated object-oriented database system and application development products which can be used separately or together. The UniSQL product suite includes the UniSQL/X Database Management System; the UniSQL/M Multidatabase System; the UniSQL/4GE Application Development Environment; interfaces to C++, C, UniSQL's Object SQL (SQL/X), Smalltalk, and ODBC; and database interfaces to Ingres, Oracle, Sybase, UniSQL/X, and EDA/SQL [ES94, Fin93, Kim94, OFAQ93, Uni91]. UniSQL/X runs on Unix workstations from Sun, HP, and IBM; the UniSQL/X client is available on Windows 3.1.

UniSQL was founded in May 1990 by Dr. Won Kim, President and CEO, delivering the UniSQL/X DBMS in March of 1992. Many of the concepts involved were developed in connection with the ORION prototype ODBMS developed at MCC in a project headed by Kim [KGBW90, KBCG89, KBC+88]. (The ORION prototype is also the basis of the ITASCA ODBMS described in Section 3.5).

UniSQL/X is a client/server DBMS that attempts to unify the relational and object-oriented data models. Access to UniSQL/X is provided by SQL/X, an optimizable database query language developed by extending ANSI SQL. UniSQL/X uses an object-oriented base, and implements a relational/object-oriented model that provides upward-compatible extensions to the conventional relational model and SQL that allow management of complex and unstructured data types. Users can build relational databases without using any of UniSQL's object-oriented features, or use these extensions, as they prefer. If only SQL89 or ODBC capabilities are used, UniSQL/X will behave like a pure relational DBMS.

4.3.1 Object Model

UniSQL/X takes the approach of generalizing the relational model in specific ways to provide object-oriented facilities. The basic generalization is that a table becomes an object class definition, and rows from that table become instances of that class. This effectively allows users to define the data type of a column to be a table, and the value of the column to be either a single row or multiple rows from that table. For example, in the table definition below [ES94]:

```
CREATE TABLE Employee
    (name        char(20),
    position     char(15),
    location     CityState,
    skills       Set-of Skills,
    employed_by  Company);
```

the value of the `Employee` table's `location` field is defined as a row (object instance) in a `CityState` table (class). This also defines a referential integrity constraint between the two tables. Similarly, the value of the `Employee` table's `skills` field is defined as a set of rows in a `Skills` table. Sets in UniSQL are not restricted to instances in tables. A set can also be a set of scalar values stored in a single column. Sets can be defined to contain unique values only, or maintained in a specified order of entry. The `Employee` table above could also have been defined using the syntax `CREATE CLASS`, with the same "column" definitions. The meaning is the same; the `CREATE TABLE` syntax is retained for compatibility with standard SQL.

The reference from the `location` field to the `CityState` table is through a unique object identifier (OID) associated with each row, and does not rely on the table having a unique key. In addition to sets, multisets (bags) and lists are also supported as aggregate values. Some of the descriptions of UniSQL/X would refer to `CityState` in the example above as a "nested table". However, the "table" is not really "nested"; the `CityState` rows referred to by `Employee` exhibit true object "reference" semantics, and can be shared with other tables. In fact, the documentation refers to "instances" or "objects", rather than "rows".

Since tables (classes) referred to by other tables are different from joined tables, it is necessary to introduce new SQL syntax to navigate these relationships. UniSQL's SQL/X language uses a *path expression* to navigate through such relationships. For example, the following query finds all employees located in California:

```
SELECT name, skill, state
FROM Employee
WHERE Employee.location.state = "California"
```

Path expressions are easier to use than joins, especially if many such references to other classes exist. However, descriptions of the product suggest that it is possible to use standard SQL outer-join syntax to perform the same functions.

When Set-of relationships are involved, a special nested cursor mechanism is used to step through the one-to-many relationship, as in:

```
DEFINE CURSOR C-Employee AS
   SELECT name, CURSOR A-Skill Skills
   FROM employee
   WHERE Skill = "COBOL"
```

`C-Employee` opens a cursor for the `Employee` table, and the `A-Skill` cursor opens a cursor on the referenced table (attribute) `Skills`.

UniSQL/X enhances the performance of navigating such table (class) references by using hashed index pointers to link referenced rows. Rather than joining tables using index searches, UniSQL uses direct pointers to navigate related rows. The use of direct pointers makes it feasible to "nest" tables many levels deep and still maintain good performance. This technique is especially important for "pointer-chasing" applications such as managing CAD data, and for scientific problems that use vectors and matrices. Recursive relationships such as bill-of-materials explosions (assembly-subassembly relationships) can also be modeled using tables in this way.

Continuing the extension of relational tables into object classes, UniSQL/X allows procedures (methods) as part of class (table) definitions. Methods are programs written in C (or any other language callable from C) which are associated (registered) with a table. Procedures can manipulate data in the associated table, access any other table, request operating system services, or call other programs and procedures. These procedures can be invoked from within SQL/X. Procedures can be used to free users from the complexity of manipulating complex data structures, or to provide special behavior for complex data types, as in any other ODBMS. An example schema from [ES94] illustrates a table with a method.

```
CREATE CLASS Library
   (card_#     integer UNIQUE,
   book#       integer UNIQUE,
   patron      string NOT NULL UNIQUE,
   overdue_amt monetary)

METHOD CLASS add_borrower(card_#, patron) integer
FUNCTION newborrow
FILE "$UNISQLX/app/bin/cpgm02.o";
```

The `METHOD` definition specifies the operator name used to invoke the procedure from within SQL/X queries, and its input and output types. The `FUNCTION` name is the name of the routine within the code that will be invoked to perform the operator. The `FILE` definition specifies the name of the file containing the actual code. In this case, the keyword `CLASS` specifies that the method is a *class method*, meaning that it operates on the class object, or on multiple instances of the class. If `CLASS` is omitted, the method is an instance method, meaning that it applies to an individual class instance. In addition to explicitly-specified methods, procedures for reading and updating the value of each defined column are implicitly available for each table. A planned enhancement involves the ability to define a column as a procedure which performs a function (e.g., computing the value of the column) when the column is referenced in a SQL/X command.

While UniSQL/X allows methods to be associated with class definitions, it does not at present (based on the limited documentation available to us) support a distinction between public and private attributes.

UniSQL/X also allows the use of inheritance to implement database subtypes. For example, the following syntax defines `Employee` as the parent (supertype) of permanent and temporary employees:

```
CREATE TABLE Employee...
CREATE TABLE Perm_emp AS SUBCLASS OF Employee...
CREATE TABLE Temp_emp AS SUBCLASS OF Employee...
```

The subtype tables automatically inherit all of the attributes (columns) of the `Employee` table, as well as having additional fields that are unique to each subtype. Methods defined for the parent table are also automatically inherited by the subclasses. Each subclass can have its own procedures, which are allowed to share the same name (overloading); e.g., the supertype and each subtype could have its own procedure called `tax_deductions` that calculates the withholding taxes for each employee. The user could invoke the `tax_deductions` procedure on a collection of instances of these types, and the appropriate calculation would be invoked automatically. Multiple inheritance is supported. All instances (rows) in the database inherit from a built-in class `Object`.

Classes and subclasses referred to from within other tables require special treatment. Path expressions that target parent tables (for example, `Employee`) also automatically search all subclasses (e.g., `Perm_emp`). It is thus possible to move row instances from the `Perm_emp` table to the `Temp_emp` table without making changes to code. In addition, UniSQL's SQL/X syntax has been extended to include an `ALL` qualification in the SQL `FROM` clause. For example, the statement

```
SELECT * FROM Employee
```

would cause UniSQL/X to search the `Employee` table, but not any of its subclasses, while

```
SELECT * FROM ALL Employee
```

would cause UniSQL/X to search subclasses of `Employee` as well as the parent table.

UniSQL/X also comes with a set of system-defined tables called *generalized large objects* (GLOs), which are organized in a hierarchical (parent/child) inheritance relationship to support a variety of multimedia data types. One leg of the hierarchy is devoted to supporting various image types, another supports different audio types, and so on. Each subtype in the GLO hierarchy has methods that can be invoked to retrieve, insert, update, and delete, as well as manipulate the multimedia data. User-defined subclasses of GLO tables, with specialized methods, can be defined to deal with custom requirements. Examples of methods on multimedia data include ones that set the `volume` or `play_time` for an instance of audio data. GLO data can be stored in UniSQL/X tables or in external files. The UniSQL Multimedia Framework also provides for seamless integration of multimedia devices such as fax machines, CD jukeboxes, satellite feeds, image compression boards, etc.

4.3.2 Architectural Features

UniSQL/X supports a client/server architecture, with the initial version providing support for multiple clients working with a single server. The architecture consists of three principal components:

- the Object Management Component is designed to address the performance demands of compute-intensive applications on memory-resident objects, and supports the management of a large number of memory-resident objects that have been transferred from the database

- the Storage Management Component includes various access methods, such as B+ tree indexing, extensible hash indexing, and signature files designed to speed up database retrieval.

- the Transaction Management Component implements a two-phase locking protocol for concurrency control, a deadlock resolution mechanism, and a log-based recovery protocol for recovery from system crashes and user-initiated transaction aborts.

The Object Management Component runs on the client machine, together with SQL/X based applications and UniSQL/4GE application development tools. The Storage Management Component runs entirely on the server. The Transaction Management Component runs on both the server and the client. In this implementation, the client process supports a significant portion of the Transaction Management component, and allows most transaction-related work to be off-loaded from the server. The client supports one process, but the UniSQL/X server supports multiple processes, one per client.

UniSQL/X applications can also be implemented in a stand-alone, single workstation environment.

UniSQL/X uses object-oriented features internally. The object storage manager and lock manager, for example, accommodate user-defined classes, complex and composite nested complex objects, as well as traditional RDBMS data types. Within the database, rows (objects) are referenced by object identifiers (Oids) instead of being referenced by a key value.

Like OODBMSs that are designed to make object-oriented programming language objects persistent, UniSQL/X provides client workspace management facilities to automatically manage a large number of objects in memory. UniSQL/X automatically converts the storage format of objects between the database format and the memory format, converts the Oids stored in objects to memory pointers when objects are loaded from the database into memory, and automatically writes objects updated in memory to the database when the transaction that updated them commits.

The UniSQL/4GE Application Development Environment provides tools which help users with schema definition, prototype development and refinement, and production application deployment via a visual editor and multimedia-style editor. The UniSQL/4GE VisualEditor tools provide users with the ability to view and edit schema information and data stored within the database. Functionally, it provides developers and end users with a GUI-based visualization of the SQL/X data definition and manipulation language. The UniSQL/4GE MediaMaster tools format the result of an SQL/X query into a report. The tools support a WYSIWYG style editor which allows end users to manipulate output and to save the resulting document as a template for reuse in other applications or for use in subsequent UniSQL/X sessions. The UniSQL/4GE ObjectMaster is a visual application development tool that allows developers, without programming, to create GUI-based applications with built-in flexible transaction management facilities, multimedia support, and integration with non-UniSQL applications, tools, and network services.

The UniSQL/M Multidatabase System enables developers to manage a collection of multi-vendor databases -- Ingres, Oracle, Sybase, DB2, UniSQL/X, and others -- as a single federated database system with full object-oriented capabilities. UniSQL/M is derived from UniSQL/X. UniSQL/M users can query and update the global database using the SQL/X database language. UniSQL/M maintains the global database as a collection of views defined over relations in attached relational databases and classes (relations) in attached UniSQL/X databases. UniSQL/M also maintains a directory of attached database relations and classes, attributes, data types, and methods, that have been integrated into the global database. Using this information, UniSQL/M translates queries and updates to equivalent local queries and updates for processing by the attached database systems. The results of these local operations are passed back to UniSQL/M for format translation, merging, and any necessary postprocessing. UniSQL/M also supports distributed transaction management over attached databases.

In September 1992, UniSQL was selected by the Petrotechnical Open Software Corporation (POSC) to provide database technology which is being used by POSC in their development of a new data management specification for the oil and gas industry. Also during 1992, because of its powerful multimedia capabilities, UniSQL was selected by the MIT AthenaMuse Consortium on multimedia as the consortium's multimedia database system.

UniSQL is pursuing a number of joint development activities with other DBMS vendors. For example, Versant and UniSQL are jointly developing a SQL/X interface layer for the

VERSANT ODBMS (see Section 3.4), which Versant will market, sell, and support ("Executive Brief", *Object Magazine*, 3(6), Feb. 1994, p.8). Cincom and UniSQL are also jointly developing technology to allow integration of Cincom's Supra relational DBMS with UniSQL/X. Initially, Cincom will remarket UniSQL/X to customers requiring an ODBMS. Later, an interface will be developed allowing users to access and update Supra data from within the UniSQL/X ODBMS. Other joint development activities between Cincom and UniSQL are also possible (*ComputerWorld*, July 25, 1994, p. 67).

4.4 Other Object/Relational Developments

In addition to the ORDBMSs described in the previous sections, some other relational systems are beginning to include what might be considered object facilities, or at least the beginnings of them. As discussed in Section 5.2, some object features are likely to be included in the next SQL standard, although it is not yet clear how extensive these features will be (and which features will be included in the next standard, versus which features will be deferred to subsequent standards). In some cases, the extensions to these relational systems may be in anticipation of these forthcoming additions to the SQL standard, in addition to their general usefulness. It is safe to say, however, that, as object-oriented features are added to the SQL standard, relational systems will implement them. Moreover, as relational DBMSs implement object features, they will approach being, if not actually be, object/relational DBMSs (depending, of course, on how "object/relational" is defined or understood).

For example, [Pir94] describes object-oriented features planned for the next version of IBM's DB2 Client/Server. These features include:

- user-defined types (UDTs)
- user-defined functions (UDFs)
- large objects (LOBs)
- constraints and triggers

As described in [Pir94], these facilities are similar to the corresponding facilities in the Object/Relational systems already described in the previous sections. For example, UDTs, together with UDFs, allow users to define new data types, which can then be used to define columns in tables, be referenced in query expressions, etc., in the same way as built-in types. UDFs are integrated into query optimization, which can take into account the semantics and execution costs of the UDFs.

Things are less clear with other relational vendors, with the exception of ASK's Ingres, which has a number of Object/Relational features already (partly based, like those in Illustra, on work done in the POSTGRES project). Relational vendors generally support BLOBs, and stored procedures, but not object identity, inheritance, encapsulation, and the full integration of user-defined types into SQL queries. Oracle and Sybase are known to have object-oriented projects in progress. Sybase is supposedly implementing a new database engine that implements object capabilities directly. For Oracle 8, Oracle is said to be integrating an object layer into its architecture, to run on top of a relational engine, rather than build a new ODBMS from the ground up (*ComputerWorld*, May 30, 1994). Oracle 8 is supposed to provide object facilities that are upward-compatible with relational facilities. Users can use either a conventional relational API, or an object API, which will provide "object windows" onto relational data (*Infoworld*, June 27, 1994, p.22).

Some of the delay by relational vendors in offering more complete object facilities may involve the vendors being unwilling to go public with specific features until the SQL3 object facilities become more solid. However, it is also likely that vendors are finding it difficult to add object features to an existing relational engine. (In some respects, this mirrors the difficulty the SQL standards groups are having adding object features to an existing relational standard). Fully-integrated object/relational databases comparable in maturity to the current RDBMS products in essence require *both* the maturity of the RDBMSs in their relational capabilities, *and* the maturity of the OODBMSs in their object capabilities. Achieving this may take some time, and may require fundamental changes to the underlying DBMS engine. Implementation issues are discussed further in Section 6.

A simpler and nearer-term approach to providing mixtures of object and relational facilities is that of building an object layer on top of a relational DBMS, as discussed above in connection with Oracle 8. This technique is used in the HP OpenODB and Odapter products discussed in Section 4.1. Of particular interest in this context is the ability of Odapter to run on top of Oracle, and possibly other (non-HP) relational DBMSs in the future. Odapter thus serves as an example of this general strategy for migrating from relational DBMSs to object facilities.

A different (but somewhat related) approach to integrating object and relational capabilities is provided by Persistence [KJA94, OFAQ93]. Persistence is an application development tool which provides object-oriented access to existing relational databases. Persistence uses an automatic code generator to convert object definitions into C++ classes which know how to read and write themselves to a relational database. By generating the methods to convert relational data into objects, Persistence saves developers from having to write these methods for each class. Applications built with Persistence can work side-by-side with legacy systems that use these relational databases.

The general architecture used by Persistence is shown in Figure 4.4.

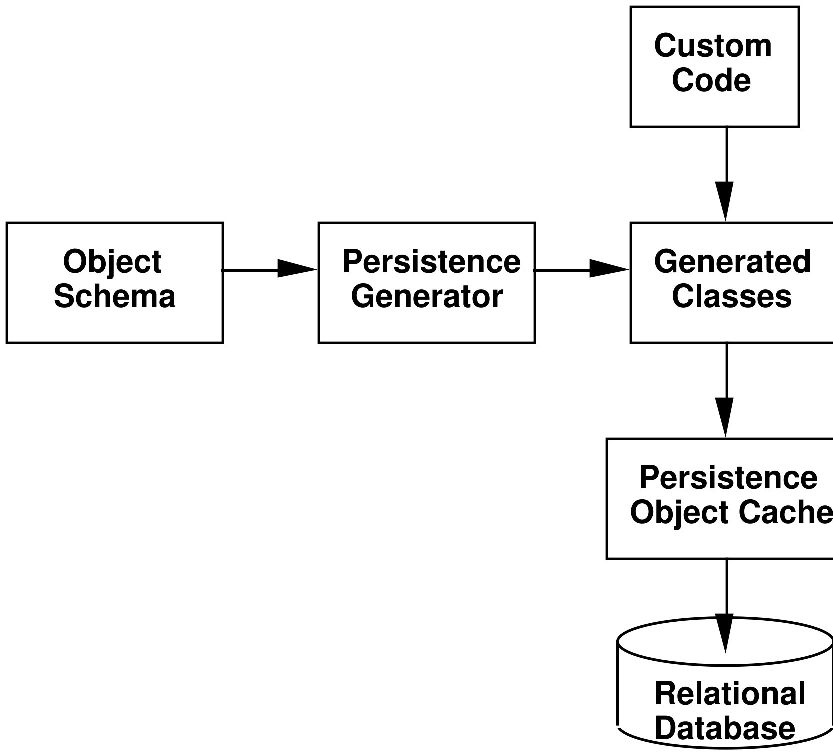


Figure 4.4 Persistence Architecture

In this architecture, the Persistence Database Interface Generator converts object schemas into C++ classes. Each class generated by Persistence maps to a table or view in the database. The user can specify the primary key attributes for each table, or ask the generator to create and maintain a unique OID for each instance. The resulting database can be queried using ANSI SQL or attribute values. Custom code can be added to the generated classes. This code can be preserved when the object model changes, and reused in the updated classes. Persistence supports inheritance of attributes, methods, and relationships in the object model (but not multiple inheritance). Superclass queries are propagated to subclasses. Virtual methods can be used for polymorphism. Persistence maps associations to foreign keys in the relational database. Each generated class has methods to access any related classes via these associations. Referential integrity between classes can be preserved using these associations (e.g., associations can have deletion constraints specified for them).

The Persistence Runtime Object Management System caches objects during transactions and ensures data integrity. In the object cache, Persistence converts (swizzles) foreign key attributes into memory pointers, which improves traversal of object references by applications. Persistence supports a number of different kinds of transactions, including read-without-locks, consistent-read, write-immediate, and write-on-commit. As data is accessed or updated by an application, the runtime system places the locks appropriate to the type of transaction on the corresponding relational tuples using the database's locking mechanism. When a transaction is committed, Persistence walks through the object cache and writes out changes to the database, and releases the necessary locks.

Persistence will support major Unix platforms (support for Intel/NT is planned for this year). Persistence supports major C++ compilers, and integrates with GE's OMTool, allowing developers to go directly from an object model to a running C++ application. Code generated by Persistence is database independent. The developer can choose from a number of relational DBMSs at link time, increasing application portability. Persistence supports Oracle, Sybase, Ingres, and Informix (planned for 1994), with ODBC support also planned for this year. Persistence also has an interface with the Objectivity/DB OODBMS (see Section 3.6), which provides a migration path to object databases.

Persistence recently joined the Object Database Management Group (ODMG), and announced that it plans to comply with the ODMG-93 specification (see Section 5.1) by mid-1995 [Per94]. [Per94] also described plans to modify Persistence to provide links between relational databases and object applications built with ODMG and Object Management Group (OMG) standards. Basically, the Persistence Database Interface Generator will be modified to read ODMG Object Definition Language (ODL) schemas. The Generator will also be modified to output classes that use OMG Object Request Broker (ORB) calls to send database requests to an ORB. On the database end, the Persistence object manager will be modified to accept messages from an ORB, rather than directly from the application. This will allow object applications to route database requests via an ORB to a remote Persistence object manager (and relational database).

Persistence product literature describes the development of cellular billing software for telecommunications applications by Cincinnati Bell Information Systems, using Persistence to provide an object-oriented interface to the Oracle relational DBMS and the Tuxedo transaction monitor. A key motivation for using Persistence was the desire to use object-oriented development in building the system, while using a relational DBMS to handle the performance requirements of the large volume of billing transactions for which the system was designed. Part of the project involved tuning Persistence specifically for Oracle. The project developers also built a utility to extract information from the Oracle data dictionary for use as the basis of their object definitions.