IDA

INSTITUTE FOR DEFENSE ANALYSES

Selecting a Persistent Data Support Environment for Object-Oriented Applications

Glen R. White

Clyde G. Roby, Task Leader

March 1998

IDA Document D-2120

Log: H 97-003808

This work was conducted under contract DASW01 94 C 0054, Task T-S5-1446, for the Defense Information Systems Agency. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

INSTITUTE FOR DEFENSE ANALYSES

Selecting a Persistent Data Support Environment for Object-Oriented Applications

Glen R. White

Clyde G. Roby, Task Leader

PREFACE

This document was prepared by the Institute for Defense Analyses (IDA) for the Software Data Architecture Engineering Division, Center for Computer Systems Engineering, Defense Information Systems Agency (DISA), under the task entitled "Common Operating Environment Architecture."

The following IDA research staff members were reviewers of this document: Dr. Edward A. Feustel, Dr. Richard J. Ivanetich, and Dr. Reginald N. Meeson.

The author gratefully acknowledges the contributions of Mr. David Diskin and Ms. Sherrie Chubin of DISA.

CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

PURPOSE AND SCOPE

This document presents a summary of the major issues that a developer of object-oriented applications should consider when selecting a persistent storage mechanism for objects. By understanding the issues presented in the paper, a software developer should be able to develop insight into the selection process and recognize the effort needed to implement an application using each of the major types of architectures available for database management systems (DBMSs).

BACKGROUND

Interest in object-oriented technology has increased as commercial products have evolved. Object-oriented technology has been used to create new development environments that encourage software engineering discipline on the life cycle of applications and databases. This discipline results in compact, reusable data and code modules that can be developed and tested in virtual isolation from the rest of the code. Extending the object-oriented paradigm into the data storage conceptual and physical design allows for consistency, less database design effort, fewer mistakes, faster delivery schedules, and lower training costs.

Standards for object data management have also evolved. The three relevant standards bodies are coordinating their efforts to provide a common database interface language for all products. Object DBMSs are consistent with the Persistent State Service specification for the Common Object Request Broker Architecture (CORBA). Several are already offering CORBA-compliant interfaces.

To select a persistent storage mechanism for an object-oriented application the developer must evaluate the application requirements, select a DBMS architecture, and select a specific product within the chosen architecture. An overview of these steps is discussed below.

EVALUATE APPLICATION REQUIRMENTS

The first step in choosing a persistent storage mechanism for object-oriented applications is to evaluate the requirements imposed by the application architecture, data structures, and behavior. An examination of the schema complexity will help to specify the nature of the DBMS interface requirement. The interface standard, the performance requirements, the platform and network capabilities, and other issues discussed in this document should be examined and prioritized.

Schema Complexity

Relational tables are ideal for simple data structures and simple queries. Complex data and complex query patterns may indicate a mismatch between an object-oriented programming environment and the relational model. Using a relational storage model in such an environment may mean excessively high development and maintenance costs and will affect performance of the client and the DBMS. The developer should estimate the size of the mapping code between an object-oriented application and a relational database model in order to estimate its impact.

Selection of Standard Interface

There are two database interface standards for objects: SQL3 and ODMG 2.0. Local policy or desire for a specific standard may eliminate the use of certain persistent storage mechanisms. SQL3 adds object mechanisms to the relational model. It is relatively simple and is common among all relational DBMSs and some object DBMSs. The ODMG standard is actually a group of standards with a set of mechanisms for each of the three object-oriented languages (C++, Smalltalk, and Java) as well as an SQL-like interface. These mechanisms are more complex than SQL3 but they are consistent with the application development language and they are portable to any DBMS supporting the ODMG standard.

SELECT DBMS ARCHITECTURE

The second step in choosing a persistent storage mechanism for object-oriented applications is to select one of the four DBMS architectures that best supports the application requirements. Each architecture provides different capabilities and is targeted for different requirements. There is no formula for selecting a persistent storage mechanism, but selection of the architecture will be based on objective evaluation of a number of interrelated issues.

- How committed is the organization to adopt object technology as the dominant software environment? Are specific DBMS interfaces required?
- How will the architecture affect performance for this application?
- How complex are the data structures that need to be stored? Is the complex data programmer-defined and subject to change as the application evolves? Is the complex data frequently moved between the application memory and the DBMS.
- How much of the current data environment of legacy applications is in relational DBMSs or files?

There are four types of DBMS architectures that support persistent state for object data. The developer should understand the strengths and weaknesses of each one as it applies to the requirements of their application. The four types are as follows:

- Object-oriented DBMSs implement data as objects in the database. Object manipulation is done the same way for all objects whether they are in the application memory or in the DBMS. This provides for transparent and tight integration between the programming language and the object-oriented DBMS. They work best for developers of applications with complex persistent data structures who primarily want a programming language interface to data.
- Object-relational DBMSs take advantage of the fact that an object-oriented model can represent a table object as a special class with relational-like behavior. The database designer can choose the most appropriate mechanism for storage. They can be used as a permanent storage environment or to incrementally convert data structures from tables to objects.
- Object-relational mapping DBMSs provide an object model interface to data stored in a separate underlying relational DBMS. The mapping software in the DBMS handles the object-tuple and tuple-object conversions needed to store object data in the relational DBMS. They can provide an object view of relational data, especially if it is easily converted.
- Relational DBMSs with object extensions are relational systems that have added mechanisms for management of specific types of objects. They do not allow the typical object-oriented programmer to store user-defined objects. New object types (class definitions) will normally be developed by the database vendors or third parties

and marketed as add-on object extensions to the core relational DBMS. They are important to existing relational DBMS applications with limited object requirements.

SELECT DBMS PRODUCT

The final step is to select a DBMS product. The developer should narrow the evaluation to a small number (one to three) of candidate DBMSs from the selected architecture with features that best meet the application requirements. Particularly important features that should be considered include:

- Standards and language support
- Access to legacy data
- Schema evolution and version control
- Method and query execution distribution
- Transaction control and locking granularity
- Security
- Distribution and size of database

An evaluation of key features of most object DBMS products is contained in the *DBMS Needs Assessment for Objects* from Barry and Associates. The developer should review the relevant features from this reference and other object DBMS references and estimate each feature's impact on satisfaction of application requirements.

A final selection of a commercial persistent storage mechanism from the few candidate products should be based on:

- Examination of other applications that use the candidate products
- Discussions with developers who are experienced with the products
- Discussions with sales and technical representatives
- Examination of available features for use in the specific application, possibly including a small prototype

1 INTRODUCTION

1.1 Purpose and Scope

This document presents a summary of the major issues that a developer of object oriented applications should consider when selecting a persistent storage mechanism. This paper is targeted to developers of object-oriented systems, therefore knowledge of the advantages of object-oriented development is assumed. It compares the types of architectures for database management systems (DBMSs) that support objects, and discusses the reasons for selecting a persistent object environment based on analysis of user requirements and product capabilities. By understanding the issues presented in the paper, a software developer should be able to develop insight into the selection process and recognize the effort needed to implement an application using each of the major types of architectures available for database management systems.

1.2 Object-oriented Programming Environments

We note that the strong trend in software development is toward increasing use of object-oriented technology, and distributed object computing. This trend has been driven by several factors:

- Development of software components for reuse (e.g., Java Beans, COM components, etc.)
- Acknowledgement of object-oriented benefits for software development and maintenance
- Rapid rise in popularity of internet/www computing and Java

Object-oriented development environments have many advantages over traditional approaches.

- The conceptual modeling environment provided by a number of object-oriented analysis and design tools provide a very high-level notation to capture even the most complex ideas in a way that can be understood by non-technical functional experts. The object-oriented development environment provides for clear communication between the developers and the users. If the system is implemented in an object-oriented language and the objects are stored in an object-oriented database, the same environment can provide support from initial systems analysis through fielding and evolution of the system.

- The ability to define the object's data structure and behavior in a self-contained module enhances maintainability of the system and promotes reuse of modules for similar domains.
- The specialization hierarchy allows the developer to have multiple objects with similar, but not identical, characteristics or behavior without including complex condition checking. For example, two similar objects may have different code to display their values.

1.3 Object DBMS

Object DBMSs have been available since the early 1980's but have been of limited interest until the last four or five years, because they were considered to be immature. They lacked the robustness, extensibility, portability, and standard interface of relational DBMSs. They were used by a few highly publicized applications with notable success. These successes generated interest but the safe alternative was to stay with a proven, large relational DBMS. There was also a fear that the relatively small object DBMS companies would not survive, leaving a developer without long-term support.

Today the major object DBMS products have proven themselves to be fully functional and reliable databases that are serving a large number of critical applications. Each of the major vendors has a long list of applications that were developed with their object-oriented products where a relational product had previously failed. They can show dramatic improvement in both performance of the delivered application and the development time for the system. The need for the ability to manipulate and store persistent objects in many applications is well recognized. This recognition is evident by the variety of products reviewed in the DBMS Needs Assessment for Objects [2] and the fact that object support is being added to the major relational systems.

The major object DBMS products are rapidly adopting standards for definition, manipulation, and query of objects. (These standards are discussed in Chapter 3 and Appendix B.) Although there are still variations from the standards, the differences are normally relatively small. Some vendors support both standard and older proprietary access methods.

Many DoD applications are being developed using object-oriented languages for the following reasons:

- Complex data modeling is required. The object DBMS products offer capabilities that can be very useful in applications with complex data for advanced data modeling, data storage, and data retrieval.
- Developers want the advanced tools that promote object-oriented techniques in software engineering. These techniques support management of application complexity, promote software reuse, and lower software maintenance through encapsulation and information hiding. All access to the object is through its public interfaces.
- Support for object-oriented technology in software design, software development, database development, and new distributed computing environments allows a common paradigm throughout the application's life cycle. Many of the tools supporting these environments have been integrated resulting in an easy-to-use and highly productive environment. It allows process modelers, data modelers, application developers to work in a common environment with effective communication with the application user.
- Today, the commercial object DBMSs are mature enough to warrant serious consideration to supplement the very capable object-oriented languages and software development environments.

1.4 DBMS Market Size

The revenue from all the relational companies in 1997 was approximately seven billion dollars. The market size for object DBMSs for 1997 was approximately $200 million [1]. The object DBMS market size is small relative to the relational DBMS market but growing at a rate of almost 50% per year. The traditional relational database vendor's rate of growth has been slowing in the last three years from 40% to 30% per year. The growth numbers show why relational vendors are eager to add object capabilities to their products.

Although the object DBMS market size is small relative to their relational competition, the market share is substantial enough to support a number of stable companies that will continue to support and improve their products.

1.5 Supporting Documentation

This paper is a summary of the major issues that a developer of object-oriented applications should consider when choosing a persistent data environment. We strongly recommend several references to aid in the comparison of architectures and commercial products. They include the DBMS Needs Assessment for Objects [2], The Object Database Handbook [3], The Object Database Standard: ODMG 2.0 [4], Object-relational Database Managers [5], and Persistent Object Service Specification [6]. All of these documents, considered together, represent a survey of commercial object-oriented database management systems. These references contain detailed information needed in choosing a commercial product for persistent data storage.

The DBMS Needs Assessment for Objects [2] is a current comparison of the features of the major commercial object DBMSs [1]. It should be used as a source document to narrow the selection of potential object DBMS products that have features required for a specific environment. The state of commercial object DBMS products is changing quickly. Since the publication of the DBMS Needs Assessment for Objects [2], several important changes have occurred. Most of the major commercial products now support the Java language bindings (e.g., ObjectStore, Versant and Gemstone). The latest release of ObjectStore has dropped support for Smalltalk but added support for Java and ActiveX. In February 1996, Informix bought Illustra and integrated some object technology into Informix. In June 1997 Oracle 8 was released with limited object support. These two systems and DB2, from IBM, have limited ability to support objects that are discussed in detail in Section 2.4.

The Object Database Handbook [3] contains further explanation of the criteria used in the DBMS Needs Assessment for Objects [2]. It is also contains good background material on object DBMS products and describes how to choose one of the products.

The Object Database Standard: ODMG 2.0 [4] captures the latest work of the Object Database Management Group (ODMG). This organization is a consortium of almost all of the ODMG vendors. The ODMG and its standards will be discussed in detail in Appendices A, B, and C.

[1] International Data Corporation provides a similar report to the DBMS Needs Assessment for Object, comparing object-oriented DBMS products. More information about the content of the report can be found at the web site at www.idcresearch.com.

Object-relational Database Managers [5] is an excellent detailed comparison of several object-relational DBMSs. It compares the architectures of several object-relational DBMS products. Also, five of the products considered in the DBMS Needs Assessment for Objects [2] are object-relational DBMSs. Section 2.2 below will examine object-relational DBMSs in detail.

Persistent Object Service Specification [6] is the first specification from the Object Management Group's (OMG) Common Object Services that is related to the persistent state of data objects. It has recently been renamed as the Persistent State Service (PSS). This paper will introduce the OMG and discuss the association of this specification with interfaces to object DBMS products. OMG issued a request for proposal for version 2.0 of the Persistent State Service in June 1997. The final submissions are due in May 1998. Four proposals [7] were submitted in November 1997 by: Fujitsu, EDS, Secant Technologies, and a group of eleven companies led by SunSoft. The reader should review these documents and Appendix B for an understanding of the PSS.

1.6 Document Organization

Chapter 2 introduces the major architectures for object DBMSs. Chapter 3 discusses the issues relevant to using standard DBMS interfaces. Chapter 4 covers the issues related to choosing one of the DBMS architectures and Chapter 5 covers the issues related to choosing one of the object DBMS products. Chapter 6 gives the conclusion and a summary of the process of choosing a persistent data support environment for object-oriented applications. Appendix A introduces the relevant standards bodies and Appendix B for discusses their standards for definition, manipulation, and query of objects. Appendix C gives examples of the standards with code examples for using the standards and discusses the association between them. Appendix D includes a list of products that are currently available for object support from three of the relational DBMS vendors. Appendix E includes the major criteria used by a DoD system for selection of an object DBMS product. Appendix F lists the current major object DBMS vendors. Appendix G provides a list of document references.

2 OBJECT DATABASE ARCHITECTURES

All DBMSs provide data persistence, query, transactions, indexing and other database functions. This paper focuses on a particular type of object DBMS that has advanced capability to support object mechanisms like identity, inheritance, polymorphism, encapsulation and especially the ability of a typical object-oriented programmer to store user-defined types. This section introduces the high-level architectures of object DBMSs and relational DBMSs that provide limited object persistence.

There are a number of products that make objects persistent from one invocation of an application to another. Some object-oriented languages provide tools to save the state of the application for later processing. In the Smalltalk environment, the state of all objects is contained in an image that can be saved to disk and restored. Although these systems have flexible object-oriented capabilities they are not database management systems. Concurrent users normally cannot share this data and there are none of the traditional data management functions of a database management system.

The major commercial systems that provide for storage of objects in some form can be grouped by their architecture. There are three major categories of object DBMSs: the object-oriented DBMS, the object-relational DBMS, and the object-relational mapping DBMS. Oracle, Informix and DB2 have added the limited capability to support objects. These companies are marketing themselves as object-relational DBMSs but we have chosen to use the term "relational DBMS with object extensions" as a separate category from object DBMS in this paper. We classify these products separately because they do not yet give the average object-oriented programmer the ability to store user-defined types. These products are discussed in detail in Section 2.4.

2.1 Object-oriented DBMS

The object-oriented DBMS is the type of object DBMS that uses an object-model for storing objects. Object-oriented DBMS products support language bindings in one or more object languages that allow the storage and retrieval of objects in the format of the client application. The object-oriented DBMS maintains caches on the client and on the server that are used to reduce the frequency of disk reads. The data structure of the object-oriented client application and the data in the caches and on the disk are the same, except for syntactic translation between the object-oriented languages (which is handled transparently by the object-oriented DBMS). Object manipulation is done the same way for all objects regardless of their location. This provides for transparent and tight integration between the programming language and the object-oriented DBMS.

The major object-oriented DBMSs are *ObjectStore* by Object Design Inc., *Versant* from Versant Object Technology, *Objectivity/DB* from Object Design Inc., *POET* from POET Software Inc., and *O2* from O2 Technology.

2.2 Object-relational DBMS

Object-relational DBMS is the second type of object DBMS. They have combined the features of relational and object-oriented systems. They also have caches on each client and the server to support efficient query response. These systems take advantage of the fact that an object-oriented model can represent a table object as a special class by system-provided class libraries. The underlying model is the object-model but it has built-in mechanisms for manipulating table objects. This additional capability provides many of the advantages and disadvantages of both systems. The application developer may choose to implement some data as tables and other data as user-defined object structures. If the application retrieves data from a table then it must also translate the result into a data structure in the object-oriented application. Most object-relational DBMSs provide an interface that is a variation of SQL, similar to the evolving SQL3, and do not provide a programming language interface.

UniSQL from UniSQL Inc. and *Odapter* from Hewlett-Packard Company are the major object-relational DBMS products.

2.3 Object-relational Mapping DBMS

Object-relational mapping DBMS is the third type of object DBMS. The products provide an object model interface to data stored in a separate underlying relational DBMS. They normally do not have independent DBMS functionality but they work with one or more of the major commercial relational DBMS products. The DBMS mapping software handles the object-tuple and tuple-object conversions needed when using a relational DBMS to store object data. It still takes time to convert during execution but the process can be tuned to be relatively efficient.

The database administrator maps the table structures to object structures as part of the data definition within the object-relational mapping product. The database administrator must also optimize the relational DBMS for the typical query patterns.

Persistence, from Persistence Software Inc., is a major commercial system in this category for C++. *Java Blend,* from Sun Microsystems, is a new object-relational mapping product for Java environments. Enterworks.com is marketing *Virtual DB* as a "middleware" package for distributed heterogeneous databases. It provides all the functionality of an object-relational mapping DBMS. It has tools for mapping between various schemas with the option to map into an object schema. Currently, it has SQL, Smalltalk and C++ interfaces. The next release will support Java, and CORBA interfaces.. *TopLink,* from The Object People, is a new object-relational mapping DBMS for Smalltalk and Java environments. The Object People have integrated *TopLink* into *GemStone* to provide a homogeneous view of Smalltalk data in the *GemStone* object-oriented DBMS and legacy data from a number of commercial relational DBMSs.

2.4 Relational DBMS with Object Extensions

In this paper, the term "relational DBMS with object extensions" is used for primarily relational systems that are adding mechanisms for management of specific types of objects. Examples of this type DBMS include *Informix* from Informix Software Inc., *Oracle 8* from Oracle Corp., and *DB2* from IBM. These products are being marketed as object-relational products but they are not included as an object-relational DBMS or an object DBMS in this paper for the following reasons:

- They do not possess key functionality of the object-relational systems in Section 2.2.
- They do not allow the typical object-oriented programmer to store user-defined types.
- They do not give the object-oriented application developer the ability to easily add a new class to the database schema.

New data types (class definitions) will more often be developed by the database vendors or third parties and marketed as add-on object extensions to the core relational DBMS. Development of these add-on products requires specialized skills, tools, and procedures that are not available to the typical application developer.

Relational DBMSs with object extensions will be important to transaction-based applications with a heavy investment in relational databases that need limited object support. Most of the legacy data is contained in relational DBMSs and DBMS conversions are expensive. The data must be moved from one DBMS to another and the interfaces to all applications that use the DBMS must be rewritten.

The initial set of object support modules is impressive and covers areas that are most needed by the non-object-oriented development community. Each product offering is included in Appendix D to give the reader a familiarity with the types of object support that these systems are currently providing.

The only interface allowed for the relational DBMS with object extensions is SQL and vendor extensions to SQL. There is no programming language interface for query of a database implemented on a relational DBMS with object extensions, as there is with many object DBMSs. If tight integration with the programming language of the application is desired then this is a definite shortcoming.

Not including the relational DBMS with object extensions in the same category as object-relational DBMSs does not mean that they are not going to be appropriate for some applications. They are of interest from both a market-share and a new technology viewpoint. If the legacy environment is built with relational DBMS products and the new object requirements can be satisfied with currently available add-on products then this technology may be the most appropriate. Many of the add-on products for relational DBMSs with object extensions are for multi-media documents, but the variety of objects that are supported with this environment is expected to grow rapidly. *Because the application developer cannot extend the object model, it is essential to understand the scope of functionality that relational DBMSs with object extensions are currently supporting.* A detailed evaluation of the add-on module would be required before committing to use one of the products.

A peripheral issue for this type of technology is the placement of the DataBlade or Data Cartridge (add-on modules that support specific objects for Informix and Oracle respectively, see Appendix D) within the system architecture. Informix allows a DataBlade to work within the DBMS kernel. This would allow improperly written DataBlades to severely reduce performance or cause a system failure. The same could be said for almost any complex SQL query but the problem is more severe when the kernel crashes. Oracle is supplying the most popular Data Cartridges and including them within the kernel. For third party Data Cartridges, Oracle has built an "isolation layer" to protect the kernel from major failures related to these cartridges. If a relational DBMS with object extensions is being considered, this issue requires further investigation for any add-on module developed by a third party.

3 STANDARDS

3.1 Standards Relevant to Object Databases

There are three standards that are relevant to any discussion of object persistence that are discussed in detail in Appendices A through C.

CORBA: The Object Management Group (OMG) is a group of 800 hardware and software vendors, system integrators, and government organizations. OMG is creating object-oriented standards for application integration and the development of distributed applications. It has defined interface specifications for their Common Object Request Broker Architecture (CORBA). Especially relevant to a discussion of object persistence are the Persistent State Service (PSS) and the Query Service (QS) specifications although several other services are also involved in data access.

ODMG 2.0: The Object Database Management Group (ODMG) is a committee representing almost all of the object-database vendors. ODMG has defined a standard called ODMG 2.0. It consists of an Object Definition Language (ODL), an Object Query Language (OQL) and three Object Manipulation Languages (OML) that are each mapped to C++, Java, and Smalltalk.

SQL3: The American National Standards Institute (ANSI) X3H2 standards committee developed the Structured Query Language, SQL-92, and is currently working on a version with object capability called SQL3.

The following points are relevant to the initial identification of a standard interface requirement to persistent object environments:

- Virtually all of the object DBMS products support database interface languages that are very close to either the ODMG or emerging SQL3 standards and are moving very fast to full compliance. They all allow significant non-standard extensions to their products that support desirable features.
- The major object DBMS products also allow most queries using SQL-92 although some require additional add-on products to support this capability.

- ODMG and ANSI X3H2 are committed to unifying the two standards in a way that an ODMG-oriented product would be interoperable with an SQL3-oriented product.
- A few of the major object DBMS products are already compatible with the CORBA standards. (See Appendix B for a discussion of the CORBA specifications and their relationship with object DBMSs, ODMG 2.0, and SQL3.)

3.2 Is a Standard DBMS Interface Important to the Application?

3.2.1 Developer View

A standard interface to all DBMSs will reduce (1) the amount of programmer training and (2) reduce the amount of software maintenance required when one DBMS product is replaced by another. Programmers in an organization that support more than one interface to DBMSs must be proficient with each interface. When a DBMS is replaced by another with a different interface, all applications that use the database must be modified for the new interface.

There are many reasons that one DBMS may have to be replaced with another. The following are a few examples:

- DBMS products may become obsolete or evolve key features that are significant enough to warrant replacement by another product.
- Standardization on a DBMS may be imposed in order to reduce training on multiple products, increase interoperability between existing DBMSs, or reduce the cost of acquisition and maintenance.

The disadvantage of using standard interfaces to DBMSs is that the developer is restricted to using only the features that have been accepted as standard. The accepted standards are typically the least common set of features supported by the majority of products. SQL has been an accepted standard for a long time and some organizations do restrict their developers to using only standard SQL-92. However, most applications use capabilities that are introduced as extensions of SQL-92 in order to take advantage of the more advanced features of the product that they have chosen. Oracle SQL has many extensions to SQL-92 and its PL/SQL (procedural language SQL) is very useful in writing code that could not be done in SQL. If the use of PL/SQL were not allowed, similar functions would have to be done via another programming language such as C. This could be done but it would require considerably more development effort.

Both relational and object DBMSs have significant extensions to their respective standards. These are often very useful features like advanced locking mechanisms and enhanced capability to distribute data across multiple platforms. Often the extensions provide a capability to optimize access to a DBMS to increase performance. Although high performance is always desirable, some applications require performance that can not be achieved without the advanced features.

The developer must examine the requirements of each application carefully and weigh the advantage of using a non-standard feature against the risk of having to significantly revise the application should the DBMS be replaced. Mandating the exclusive use of standard interfaces may result in unacceptable performance and significantly increased cost of initial system development. The code that results from prohibiting the use of non-standard features, like PL/SQL, will likely be more complex than the code using the extensions, resulting in increased maintenance cost.

Another issue is whether the developer will use embedded queries in a standard query language or extensions to the native language to access data from the DBMS. Relational DBMSs only allow embedded SQL to access data. The object DBMS products support query and object definition languages, but typically the developers use C++, Smalltalk, or Java interfaces to the databases. This issue is discussed in depth in Appendix B Section B.7.

3.2.2 Query View

The most common pattern for system development is for a group of applications to use one physical database for the majority of their data storage and retrieval. Some of the data in this database may be copies of data from other sources, including reference tables or list of transactions from other systems, but this data is managed by one DBMS and stored on one server. Frequently one developer is responsible for the primary database and all the applications that use it as their primary source of data. Other applications may query data from this database but their authorization to write is very restricted, such as submission of some type of transaction.

The discussion in the previous section about using non-standard extensions to the database interface languages, was primarily applicable to systems that have frequent interaction with this primary database. Systems that have less intense requirement for data access may find that using only features of the standard language may be acceptable both for the primary database and for other data sources. Even systems with heavy database requirements for their primary database may have lower requirements to access data needed from alternative data sources. If necessary, the auxiliary data may be replicated in the primary database to reduce network delays and allow the use of familiar extensions in the local environment.

Systems, such as executive information systems and decision support systems, that require query of data from many sources will have much more need for adherence to standard database interfaces. Development of other applications that use one database for the bulk of their data requirements should at least focus on standard interfaces for alternative data sources. Since the alternative data sources are typically managed by other organizations, there is a high probability of diversity and change in these systems. Therefore, adherence to standard interfaces for the alternative sources is highly desirable to shield the developer from the underlying structural changes.

4 HOW TO CHOOSE A DBMS ARCHITECTURE

DBMSs provide multiple concurrent users of a database a consistent access to the database. DBMSs have varying capabilities for management of data, system backup and recovery, data integrity, database administration, data distribution, security, and *ad hoc* query. Application developers must evaluate their requirements against the set of capabilities in each product before selecting one.

Selecting a type of product from one of the DBMS architectures will be primarily a function of a small number of issues.

- How committed is the organization to adopt object technology as the dominant software development strategy? Are specific DBMS interfaces required?
- How will the architecture affect performance for this application?
- How complex is the current client data structure that needs to be persistent? Is the complex data programmer-defined and subject to change as the application evolves?
- How much of the current data environment of legacy applications is in relational DBMSs or files?

4.1 Consistency in Software and Database Development Environments

Object-oriented programming environments have tools that help impose a software engineering discipline on the life cycle of applications and databases. This discipline results in compact, reusable data and code modules that can be developed and tested in virtual isolation from the rest of the code. Extending the object-oriented paradigm into the data storage conceptual and physical design allows for consistency, less database design effort, fewer mistakes, faster delivery schedules, and lower training costs. Of course, these results will vary in each organization. But after the initial training period, organizations that use both object-oriented programming and database environments have shown consistent decrease in the level of effort to develop and maintain the application to database interface.

4.1.1 Consistency for the Application Programmer

Object DBMSs provide a high degree of consistency for the application programmer between the application and the database. An object-oriented view of the database schema that matches the application's object structures reduces impedance mismatch. Consistency between the application's object-oriented programming language and the database interface language is also important. Most object DBMSs allow C++, Java, or Smalltalk language mechanisms to interact with the database so the programmer only uses one language for everything. Virtually all object DBMSs support ODMG OQL and/or SQL with object extensions that may be used where a standard query language is desired. Figure 1 shows the support for database language interfaces for the major architecture types.

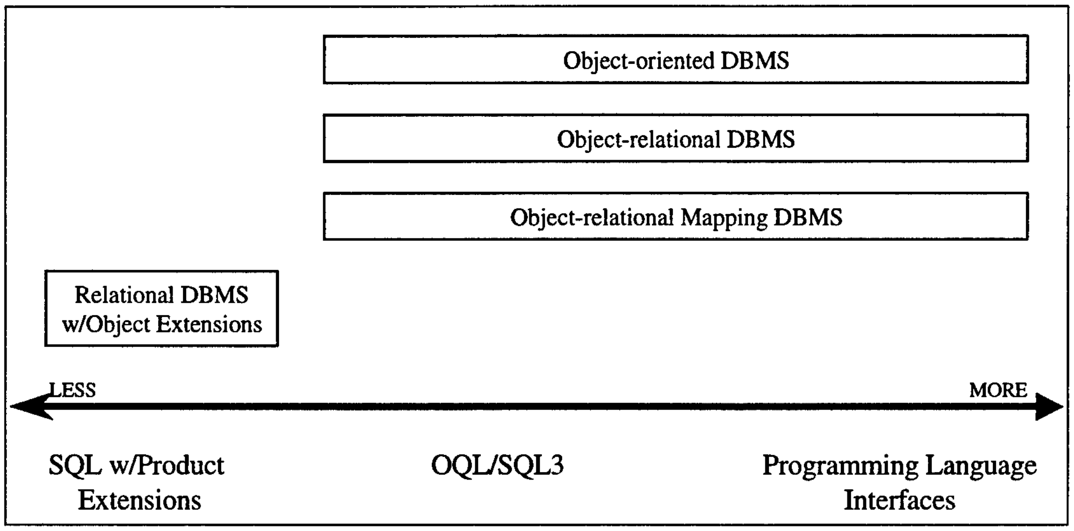

Object-oriented DBMS
Object-relational DBMS
Object-relational Mapping DBMS
Relational DBMS w/Object Extensions
LESS
MORE
SQL w/Product Extensions
OQL/SQL3
Programming Language Interfaces

Figure 1. Consistent Database Language Interfaces

Object-oriented DBMSs and Object-relational mapping DBMSs always provide at least one programming language interface to the database and most also provide an OQL or SQL3 query language interface. Object-relational DBMSs provide at least a variant of SQL and most also provide at least one programming language database interface. Relational DBMSs with object extensions provide a variant of SQL with specific extensions that support the object classes supported by the add-on module.

4.1.2 Consistency for the Database Developer

The developer should evaluate the environment for the database designer to build a database that will support one or more object-oriented applications. Figure 2 shows the types of structures that may be implemented in each of the major database architectures.

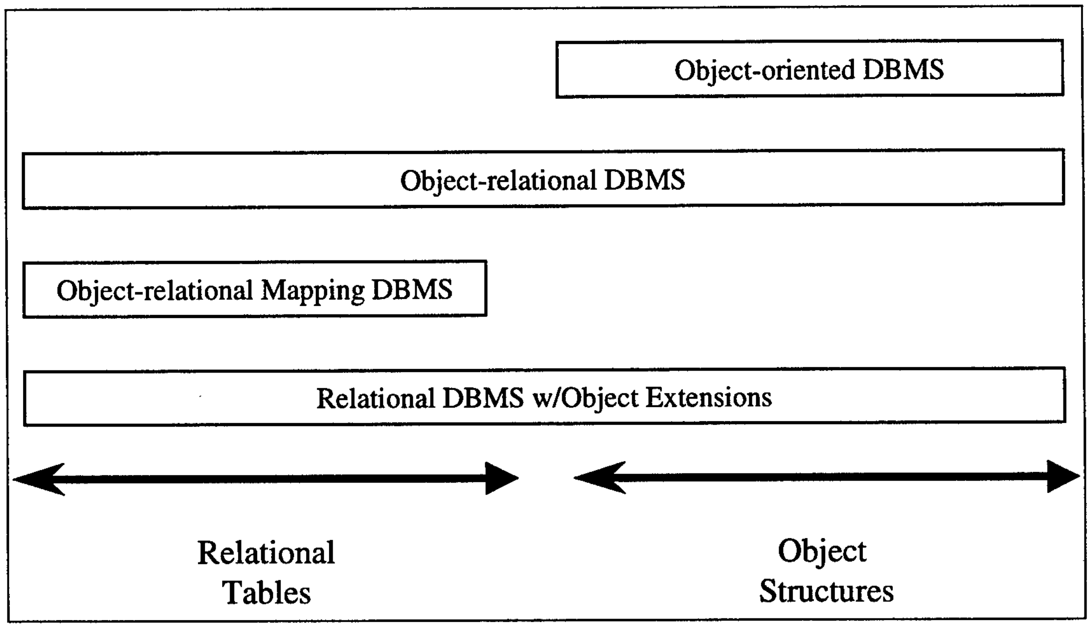


Figure 2. Database Implementation Structures

Object-oriented DBMSs only implement objects in the database and object-relational mapping DBMSs only implement relational tables. The object-relational DBMSs and relational DBMSs with object extensions may implement either object structures or relational tables.

Object-relational Mapping DBMSs require the database developer to design and implement the relational tables, and an object-oriented meta schema, and a mapping between them. Some object-relational mapping DBMSs provide automated tools to assist in the mapping process (e.g., Java Blend). It is often a difficult task to have a relational model that is tuned for query performance and a meta schema that mirrors the object structures in the object-oriented applications. Any change to either the relational or meta schema must be reflected in the mapping.

Object-relational DBMSs advocate giving the developer the option to implement the most appropriate structure for each data requirement. More than one option is good for tuning the system performance but the developer must be skilled in each option. The database developer may use a SQL variant or a programming language interface to define and create object structures. Creation of tables is done only via a variant of SQL. Mapping from one type of implementation to another type of query view within a object-relational DBMS is seldom done. If data is stored as an object, then the applications retrieve objects. If data is stored as a table, then the applications retrieve lists.

The schema development environment for a relational DBMS with object-extensions is much different for creation of a new table than for development of a new object type. The development environment for the add-on object modules requires specific software and training. Creation of a new table is a simple SQL statement.

When database schema design and the application development is done by the same group of people then consistency of the environments can be important to productivity. It is also important when frequent changes to the database schema or meta schema is required to support changing requirements of the applications that use the database. Consistency between the application and database data models and the two development environments is very desirable. Inconsistency in the application and database schema is the issue in the following section.

4.2 Impedance Mismatch

When there is a difference between the data structures used in the object-oriented application and the data structures stored in the database the data must be converted on each read or write. If object-oriented data structures are more complex than simple record structures that can be captured in rows of tables then the translation code will also be complex. The difference between the application's object-oriented model and the relational DBMS's schema is known as "impedance mismatch." An illustration of this situation is in Figure 3. The more complex the client data structure is, the more effort is involved with translating to and from the relational tables. Representing object data in relational tables is normally possible but it adds significant overhead, especially if the translation is done by the client application.

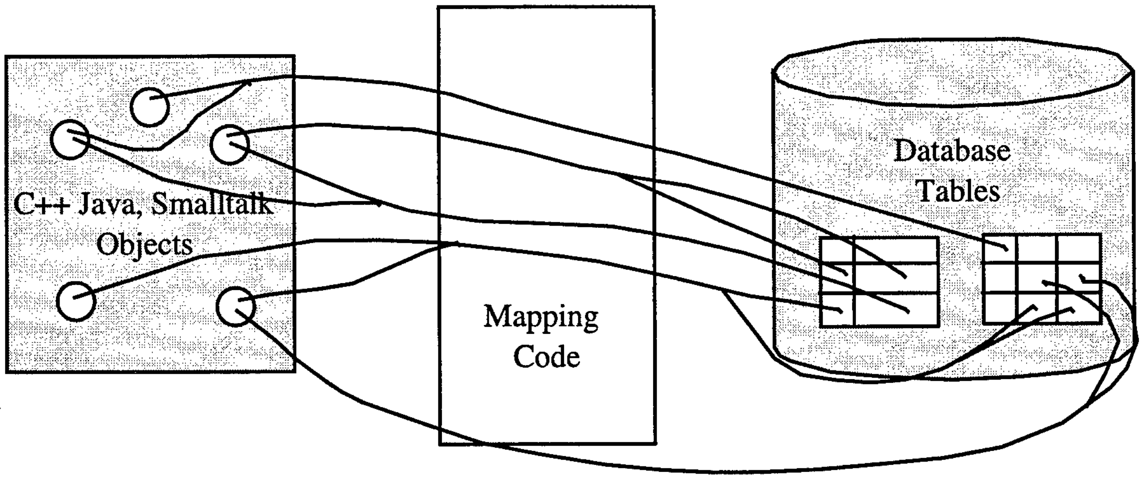

C++ Java, Smalltalk Objects
Mapping Code
Database Tables
Application Data Structures
Application Code or Object Relational Mapping DBMS
Relational DBMS

Figure 3. Impedance Mismatch

The problems with the translation and transfer of data from one environment to another are (1) that it takes time during execution and (2) the programmer must develop and maintain the translation code.

The major feature of the object environment is the rich object modeling capability that allows clear understanding of the data and evolution of the behavior and associations between objects. When translations must be made to other environments then an artificial, less-meaningful representation must be used by the applications that use the database. The semantic meaning of the data must be translated correctly. This takes time and additional table structures to represent the complex relationships of the object model.

In typical object-oriented applications with moderately complex data structures and database access requirements, the amount of code that is dedicated to accessing a relational DBMS and translating to and from the relational model, may be approximately 40% of the application code. Using an object DBMS for those applications may reduce the database access code to 10% because there is no translation required by the application. For example, consider a one million lines of code application in C++ using a relational DBMS. Approximately 400,000 lines would be dedicated to query and translation of the lists into object structures. By using an object DBMS, with a C++ or OQL interface, the number of lines could be reduced to 100,000. Therefore, 300,000 lines of code need not be developed or maintained.

In object-oriented DBMSs no mapping code is required. In object-relational DBMSs no mapping is required for data stored as objects, but the application must contain mapping code for any data retrieved from tables that is held in application object structures. In object-relational mapping DBMSs the database developer designs the mapping code, which executes on the database server. The application queries objects.

Given that complex data structures result in impedance mismatch between application objects and relational tables, a critical step in analysis of the application requirement is to analyze the complexity of a schema.

4.2.1 Analysis of Schema Complexity

The first indicator of the level of schema complexity will be the number and cardinality of associations between objects. If those associations are many-to-many (each thing is associated with many other things) then the schema is more complex.

Many-to-many relationships must be "normalized" away in a relational implementation. This normalization adds a third association table to represent two one-to-many associations between the two tables. Queries to the set of three tables are even more complex because the association tables and the tables that they associate have to be joined for every query involving data from the two original tables. The effect of increased level of complexity of the definition code is demonstrated in the Section C.8.3 in Appendix C. This type of normalization is not required when the data is implemented in an object model.

Draw a graph of the schema with lines drawn between the objects that are related. Complex data will have a very strongly connected graph with complex associations. An example of a graph of this type is contained in Figure 7 in Appendix C.

Whenever the size, type, or cardinality of a field cannot be predicted, the data is more complex. Complex data frequently contains uniquely formatted data such as sound or pictures that do not fit in fixed size and format slots. If an attribute may be of variable size or type then a class hierarchy may be defined to capture the variance between the types of objects. This is much simpler and more efficient to implement using objects than relational tables.

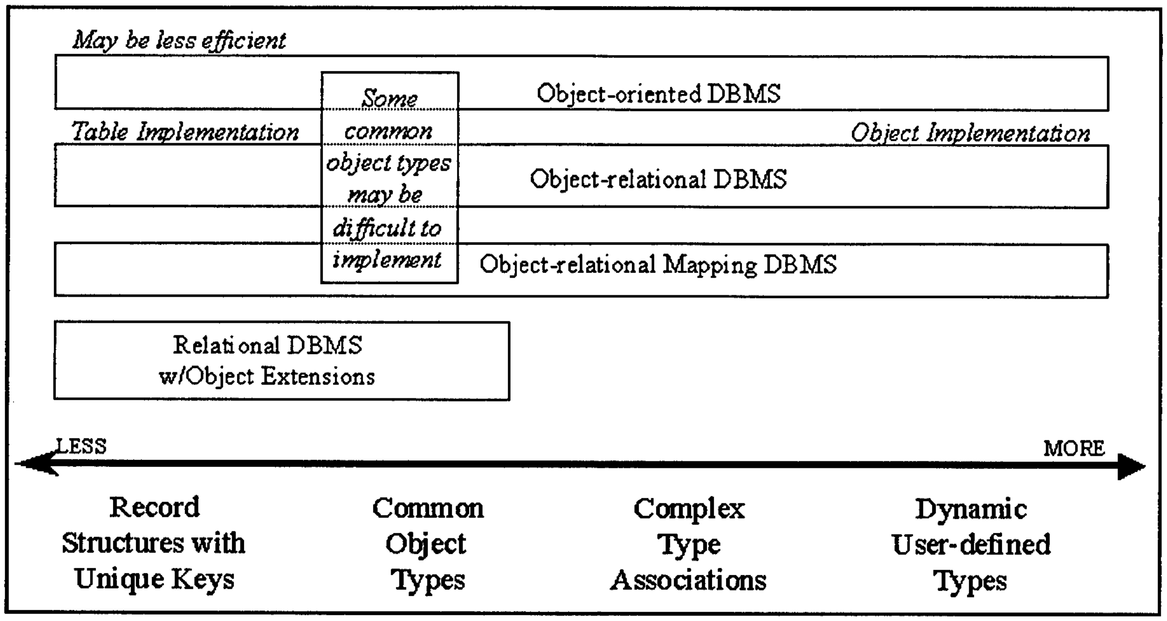
If a variable may have more than one value, then it must be "normalized" for a relational database. The "normalization" process breaks down logically associated data into multiple tables to ensure that each value of each row has only one value. This is also not required if the data is implemented as an object.

In a relational system, data elements from different tables are associated and collected by comparing the values of columns (fields). Each table must have one or more columns that uniquely identify each row in the table, known as the primary key. If there is not an obvious unique set of fields for the primary key then an artificial key must be assigned to each row, making the data model more complex. Since the object model does not require a unique key, this is not required for an object model implementation.

Type codes are sometimes used in a relational schema to indicate membership in a group with certain characteristics. An example would be a seniority column in a table, associated with a manager, that would indicate "senior" or "junior" manager. This field might be used by code to assign senior managers differently than junior managers. The presence of these type codes is another indicator of complex data. This might be modeled in an object-oriented model as a specialization hierarchy with senior_manager and junior_manager as subtypes of manager. Each would have different methods for assignment of its managers and no type code would be needed.

Complex data is frequently present in engineering design and manufacturing systems, logistics, financial portfolio risk analysis systems, telecommunications service applications, World Wide Web applications, multi-media document structures, and hospital patient record systems. Another good example of a complex data model can be found in chapters 2 and 3 of *The Object Database Handbook*[3].

Figure 4 illustrates where the major DBMS architectures are most suitable for various types of schema complexity. Any of the object DBMSs may support any schema type. But they may not be as efficient as relational DBMSs if virtually all of the data is simple record structures. The object-relational DBMSs have the option to implement either tables or objects. Therefore, they have the potential to compete with any of the other architectures. The object-relational mapping DBMSs implement as tables but give the client applications an object-oriented view of the data. If the schema is complex the mapping will be complex. Although the application is not concerned with the complexity of the mapping, the mapping code must be executed and the tables must be joined for each query. When a relational database is used primarily by non-object applications, object-oriented applications may use an object-relational mapping DBMS to view the data as an object. These objects may be associated with objects derived from other databases. These DBMSs are well suited for providing an object view of relational databases that are relatively easy to map to objects. Relational DBMSs of all types do not easily support complex and user-defined data. The new systems with add-on object extensions are ideal for objects that are used in a common and static way by many applications. Some common object types, like multi-media documents, will have a wide customer base. Other common, but complex types, like specialized images such as "face recognition," may require very detailed methods to query and categorize the images. These may be provided by relational DBMSs with object extensions but will not be commercially available to other DBMS architectures. Development of a similar capability by an application developer may be difficult and therefore not economically feasible without a wide customer base.



May be less efficient
Some common object types may be difficult to implement
Object-oriented DBMS
Table Implementation
Object Implementation
Object-relational DBMS
Object-relational Mapping DBMS
Relational DBMS w/Object Extensions
LESS
MORE
Record Structures with Unique Keys
Common Object Types
Complex Type Associations
Dynamic User-defined Types

Figure 4. Schema Complexity

4.2.2 Analysis of Query Patterns

The identification of complex data in the data model is not necessarily a sufficient reason to justify the use of an object DBMS. Further analysis should be made of the query patterns of the primary client applications. If most of the queries are asking for information about a small number of data structures (objects or tables) then a relational system may be the most appropriate. An example is a query to a military personnel database for the names of the Intelligence officers at Ft. Riley Kansas in the grade of O-5. Another example is from the baseball application (Appendix C). If most queries are for lists of players with a particular position or team, where the data is contained in one or two tables, then the relational model may be adequate because the mapping code will be relatively simple.

Also, if the application reads data from the database, converts it to object structures, and then retains it in memory for long periods the processing overhead to execute the mapping code is relatively low.

If a typical query requires frequent data from three or more tables, then queries are probably complex and the use of an object DBMS may be warranted. If a typical query to the military personnel database gets all of the performance evaluations, dependent data, picture, skill qualifications, etc., for one officer then an object database query may be the most efficient. In the baseball example (Appendix C), queries about the effect of a game's ticket sales on that game's batting statistics (to see if large crowds motivate batters) may involve four to six tables with very complex join conditions. The same query in an object DBMS could be contained in a single query using path traversals along object associations and invocation of methods contained in the Game object to calculate game statistics.

Since large table joins can slow database queries greatly, many developers replace them with a series of simpler queries. This may be a series of queries where the results of one query are used in a subsequent query. A similar form of query is the nested query where the result of one query is used as an internal component of another query (normally within parentheses). Series of queries and nested queries are substitutes for table joins that cumulatively require just as much or more processing overhead as a complex join. They are a good indication of a complex query pattern. A series of queries of this type should be evaluated as if it were a single query for evaluating query patterns. An example of a nested query is located in Figure 31in Appendix C.

The heuristic of three or more table joins, as an indicator of complex queries is not sufficient by itself. The number of rows in a table (or number of instances of an object) that are involved in the complex join is the major performance driver in executing the complex query. Actual implementations of table joins are very specific to each relational DBMS vendor and a highly optimized mechanism is essential to a successful commercial product. Conceptually any relational DBMS must compare the cross product of each row in all the joined tables to see if the join fields match. If one or more of the tables in a four-table join have ten rows then they have a small impact compared to joining with a table with 100,000 rows. Joining with only two of these large tables can be very slow. The "three or more" heuristic assumes an average size of the tables being joined, which will vary between applications, but for this discussion consider 500 to 1000 rows to be average. The more rows that actually match during a join and the number of join conditions also increase the processing load during a join operation.

Any DBMS would have similar overhead finding an object in an extent of 100,000 objects or a row in a table of 100,000 rows. Systems of both types may have indexes and other search mechanisms to accomplish this, although the relational DBMS mechanisms are frequently more optimized for this type operation. For the object-oriented DBMS, once the desired object is located, retrieving the associated data in other classes of objects are simple path traversals rather than complex joins on large tables.

Figure 5 illustrates where the major DBMS architectures are most suitable for applications with various types of query patterns. Any of the object DBMSs may support any query pattern. They may not be as efficient as relational DBMSs if virtually all of the queries are storage and retrieval of transactions that can be represented as simple record structures. Object-oriented DBMSs and object-relational DBMSs accessing their own object structures are best at queries that require accessing many object types in the same query. The object-relational DBMSs have the option to implement either tables or objects. They will have behavior similar to relational DBMSs for objects implemented as tables. The object-relational mapping DBMSs implement as tables but give the client applications an object-oriented view of the data. If the query patterns require access to many objects the mapping code may be complex or require excessive joining of tables. Relational DBMSs of all types can support combining data from many tables but will almost always be very inefficient.

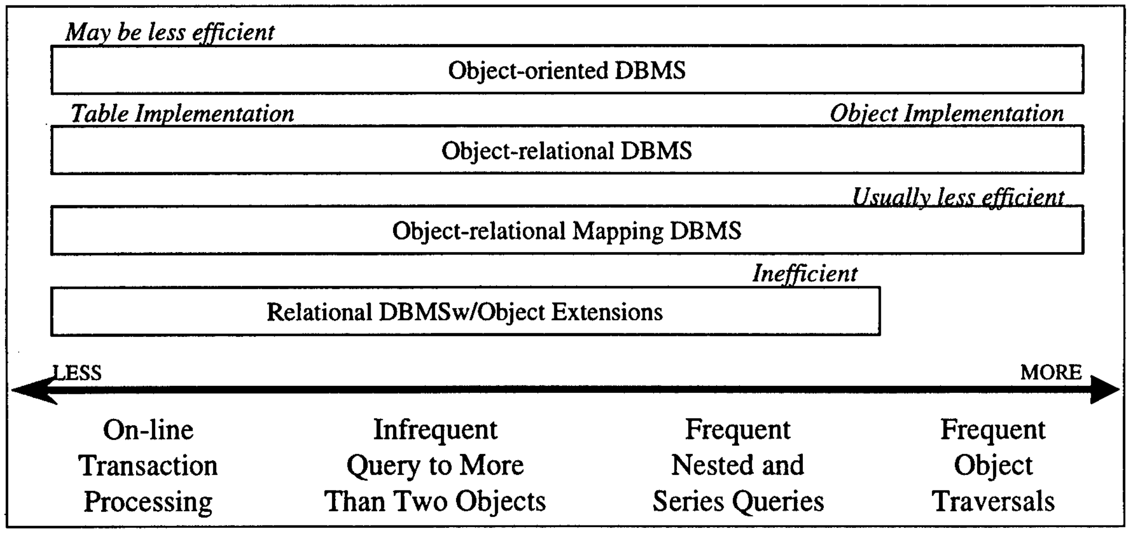

May be less efficient
Object-oriented DBMS
Table Implementation
Object Implementation
Object-relational DBMS
Usually less efficient
Object-relational Mapping DBMS
Inefficient
Relational DBMSw/Object Extensions
LESS
MORE
On-line Transaction Processing
Infrequent Query to More Than Two Objects
Frequent Nested and Series Queries
Frequent Object Traversals

Figure 5. Complex Query Patterns

4.2.3 Estimate Size of Mapping Code

Once a significant impedance mismatch problem has been identified the magnitude of the additional coding for translation between the two data structures should be estimated. The estimate of 40% of relational DBMS query and translation code versus 10% object DBMS query code in Section 4.2 is based on a number of examples. A better estimate for the specific application should be made in order to analyze the alternatives.

- Consider a representative sample of queries (any database interface code segment that reads or writes to the database) that will be executed frequently against fairly complex data in the application data model.
- Write the queries against both object and relational databases and compare them. This will give a measure of how much code will be involved in the translation between the relational and object models.
- Estimate the frequency of these queries to give a rough estimate of performance because the translation code must be executed each time a read or write is used.

The primary issue is the amount of additional code that must be written and maintained. An estimate of 30% of the code is devoted to addressing the impedance mismatch for object-oriented applications with a moderately complex schema using a relational DBMS. Performance will also suffer if there is a lot of additional code that must be executed frequently or if the DBMS must join many tables for frequent queries.

4.3 Performance and Transaction Models

Performance issues are often critical to the selection of a DBMS but *ad hoc* decision criteria based on performance are difficult to specify. Performance requirements routinely dictate server size and communications bandwidth. Analysis of the transaction model will help to focus on the most likely DBMS product for the best performance for persistent storage and query.

There is very little benchmark data that allows comparison of DBMS products from different architectures on an equal footing. Relational DBMSs have traditionally used the TPC-C benchmark for transaction processing systems and have even tailored their products so that they would perform well on those benchmarks.

TPC-C benchmarks are good for comparing different relational DBMS products but object DBMS vendors have not published TPC-C results. The TPC-C benchmark is an on-line transaction processing benchmark with multiple transaction types. It involves a mix of five concurrent transactions of different complexity; either executed on-line or queued for deferred execution. The database is composed of nine types of records with a wide range of record and population sizes.

Object DBMS vendors have published 001 and 007 benchmarks that are tailored for applications with complex retrievals that require traversing many links between objects. The 007 benchmark is newer than the 001. It measures speed of many kinds of pointer traversals, including traversals over cached data, traversals over disk-resident data, sparse traversals, and dense traversals. It measures updates to indexed and unindexed object fields, repeated updates, sparse updates, updates of cached data and the creation and deletion of objects. The relational DBMS products have not published results on this type of benchmark.

Performance was an issue during a panel discussion of "Database Trends" at the "Enterprise Outlook Conference" on June 30, 1997. The panel members were David Banks, President and CEO, Versant Object Technology; Robert Goldman, President and CEO, Object Design; Jerry Held, Sr. Vice President of Server Technologies, Oracle Corporation; and Michael Stonebraker, Vice President and CTO, Informix. Michael Stonebraker said,

> "There's 001 and 007, that are basically object benchmarks. Those guys (object DBMSs) run great on those benchmarks. These guys (relational DBMSs with object extensions) lay an egg. On TPC-A and TPC-C, these guys (relational DBMSs with object extensions) run great and those guys (object DBMSs) lay an egg. On very different kinds of applications, there's no question that there's a class of applications that this side of the platform works best on. There's a class of applications that that side of the platform works best on."

There was a virtual consensus among this group of experts that relational DBMSs are better at TPC-C type of problems and object DBMSs are better at 007 type of problems could not be argued. Transaction-based applications require simple queries of and process lists that closely match a table row. These client applications do not have a requirement to query or manipulate complex data relationships. Therefore fast simple table operations are optimal for relational DBMSs. Likewise, client applications that require heavy use of traversal between objects (equivalent to relational table joins) would have better performance on a object DBMS.

There are successful applications running on object DBMSs that are transaction-based but the developers made a decision to move to object DBMSs based on other factors, including satisfactory performance. The applications that tend to have performance problems on relational DBMSs are those that have to join many tables and map row structures into complex data structures in memory.

Object-relational mapping DBMSs have many applications that claim to have high performance requirements. When a client application uses a complex meta schema that must be mapped to joined tables, there is an inherent inefficiency. However, the systems have optimized the mapping. The main advantage is that the application developer does not develop and maintain the mapping code for each application. The other difference is that the database server executes the mapping code rather than the client.

4.4 Access to Legacy Data

There are currently massive legacy data stores in DoD. Migration of this legacy data to an object database would be expensive. Some applications may be best suited to access the data in a relational format but others would prefer to store and retrieve objects. Access to at least some of this legacy relational data is probably a requirement for most applications. When many applications use a database, changes to the database must be coordinated among all users. Determining the optimum time to convert a common database to an object DBMS requires analysis for each system to examine costs and benefits of each option. For each option, total the result of the analysis for each affected system to make a business-case for the decision.

The relational DBMSs with object extensions are the most adept at accessing legacy relational data. Oracle, Informix, and DB2 use the structures that have always existed in their relational model. Relational DBMSs with object extensions have some ability to represent object data that is derived from tables, like the object-relational mapping DBMS products, but this new capability only works with the internal relational DBMS. Relational DBMSs with object extensions either represent data as tables or as specific types of objects supported by add-on modules. A user with almost all legacy data in a relational DBMS and minimal object-oriented code should examine the object requirement carefully. If one of the commercial object modules of a relational DBMS with object extensions will support the requirement (e.g., multi-media documents) then a relational DBMS with object extensions might be the best choice. Performance is not likely to be an issue in choosing this architecture assuming that there is no highly complex data being manipulated by the applications, otherwise there would be some object-oriented code development.

Object-relational mapping DBMSs are designed to extract data from legacy systems and present it as objects to object-oriented applications with the translation performed by the DBMS mapping code. Since they do not have internal relational DBMS support, they focus on integration with the major relational DBMS products. If the primary requirement is to provide object access to a few new object applications, this type of system might be a wise choice. They are generally most desirable as interim object-support to multi-application databases until the number of object-oriented applications warrant conversion of the underlying DBMS.

Object-relational DBMSs also provide a mechanism for accessing legacy data. For best performance, the data may be transferred into the internal relational tables provided within the object-relational DBMS. This type of product has the capability for distributed heterogeneous access to data. UniSQL provides the capability to retrieve data from most of the commercial object DBMS products and present it as if it were internal data. It also has similar add-on capability to access most of the commercial relational DBMS products. Providing a unified view of heterogeneous data sources is not an easy task with any tool because of the differences in semantics and structure of data. The distributed data access tools available with object-relational DBMSs (and to a lesser extent with object-oriented DBMSs) provide a database-centered mechanism that relieves the client application from the differences in structure or location of data.

Object-oriented DBMS products do not have as much native capability to access data from different database products but there is significant capability provided by add-on routines to import data into its environment. ObjectStore has a DBconnect product that is specifically designed to extract data from non-object systems when required and present it as if it were stored locally. Ontos has specialized in creating interfaces from object DBMS products to relational systems either during run-time or during the data conversion process. The DBMS Needs Assessment for Objects [2] contains a section on external DBMS access.

5 HOW TO SELECT A DBMS PRODUCT

This chapter discusses issues related to selecting a DBMS product after the choice of architecture has been made. In addition to the issues in Section 1, the issues in this chapter should be examined closely in a comparison of the application requirements with product capability. Appendix E examines an example decision process used by the US Transportation Command (USTRANSCOM) in making an object DBMS product selection. This chapter focuses on object DBMSs but most issues are also relevant to selecting one of the relational products.

Just as there is no formula to use for making the DBMS architecture selection, there is no generic formula for choosing a DBMS product from that architecture. The developer should narrow the evaluation to a small number (one to three) of candidate DBMSs from the selected architecture with features that best meet the application requirements. Some of the features may not be relevant to selecting an architecture but are important when selecting a specific product. For example, distribution is not an important feature in selecting an architecture because there are products within each category that have powerful distribution capability as well as products with minimal distribution capability. Particular important features for selection of a DBMS product include:

- Standards and language support
- Access to legacy data
- Schema evolution and version control
- Method and query execution distribution
- Transaction control and locking granularity
- Security
- Distribution and size of database

An evaluation of key features of most object DBMS products is contained in the DBMS Needs Assessment for Objects [2]. The developer should review the relevant features from this reference and other object DBMS references and estimate each feature's impact on satisfaction of application requirements. A discussion of features not previously discussed follows.

5.1 Schema Evolution and Version Control

Schema evolution and version control are among the features that can simplify the programmer development effort in some applications. Powerful schema evolution capability can greatly reduce complexity in applications with a schema that evolves as programmers define new object capability. Some DBMSs require creating a separate new schema and importing data from the old schema. Others allow dynamic changes even while applications are accessing the database.

Version control allows the programmer to capture the state of one or more groups of objects as a named version. Applications that develop and evolve complex objects, such as engineering designs and planning systems, may benefit from this feature. Some DBMSs allow multiple versions to be accessed concurrently. Some allow versions to be restored and accessed one at a time. Others do not allow explicit identification of versions.

5.2 Method Execution Distribution

Object methods may execute either in the server or on the client workspace. The distinction may make a difference in the performance of some applications. Execution is more than just balancing the workload between the available processors. Most of the data resides on the database server. Methods that operate on a lot of data in order to produce a relatively small result are likely to perform better on a system that allows method execution on the server. Execution on the client requires transfer of all of the necessary data from the server for the query.

Conversely, methods that operate primarily on data that is otherwise needed in the client's workspace, especially if it is executed repeatedly, will run faster if executed on the client. Of course, methods that operate on the client are greatly enhanced by the presence of a database cache on the client to reduce the frequency of data retrieval.

All of the DBMS products execute some code on the server to support triggers, consistency and constraint checking, error reporting, etc. Most systems allow execution of limited user-defined code segments on the database server. Smalltalk-based DBMSs, like GemStone, are active databases. Active databases allow an application to store a user-defined method in the database and execute it immediately. The ability to support an active database is enhanced by the ability to store and retrieve Java objects in object-oriented DBMSs and some object-relational DBMSs. An application may execute Java-based methods on the server or retrieve an object from a database and execute its methods on the client. A very useful example of this is downloading an object containing its own data viewer specifically designed for the type of data contained in the object.

5.3 Query Execution Distribution

Location of query execution is a similar issue to method execution. Queries executed on large amounts of data are more likely to process faster on the server by avoiding transferring large amounts of data. Queries that execute on data that is already primarily in the client workspace will execute faster on the client. Queries may be split between the client and the server by the developer's design. For example, in the baseball application (Appendix C), the server may execute a query using an index on Players to locate the set of Players who pitched winning Games at a particular Stadium. The Player objects could be transferred to the client cache where additional queries can be executed on the collection. The developer should consider the support provided by the object DBMS for this capability in the product decision.

5.4 Transaction Control and Locking Granularity

The ultimate objective of a locking scheme is to restrict as little data as possible for as short a period as possible and still protect the integrity of the database without affecting performance. However locking mechanisms require overhead during run-time and are complex to build into in a DBMS. Some products are designed to swap pages into and out of the user address space quickly with locks on all of the data being viewed by the user. This is satisfactory for systems with high performance requirements and few concurrent users. Most systems provide many options in the granularity of locks. The designer is able to select the most appropriate mechanism. Object DBMSs provide options for locking granularity (e.g., page, index, segment, file, object clusters, containers, single object types/classes, single object instances, and single attributes of single instances). The developer must again evaluate the application requirements and select a product that satisfies the requirement.

Control of transactions is a similar issue. Long transactions make local changes to data without publishing the changes to the database so that other users have access to the changes. Write locks on this data must be held until the transaction is complete and others may have to wait until the lock is released. Short transactions have more frequent synchronization with the other database users with a corresponding increase in overhead for more frequent reads and writes to the database. The important thing in choosing a DBMS product is flexible support for the type of transaction controls that may be required by the target applications.

5.5 Security

The security mechanisms in DBMSs vary widely. Some provide security only at the database access level, giving all-or-nothing access to the database. Others provide detailed access controls at the file, partition, segment, schema, class, instance, method invocation, attribute, or even value of attribute level. The most capable relational DBMSs provide access controls down to the row level but they are infrequently used. Most applications use only table (equivalent to object) level locking. The security mechanisms are quickly being enhanced by object DBMSs, so a current study of the security capabilities should be made after consulting the DBMS Needs Assessment for Objects [2].

None of the object DBMSs have successfully undergone DoD security certification. The certification process is too costly and takes too long for a relatively small company to justify spending the resources necessary to complete the certification. Their products are changing rapidly and the certification process must be repeated for each version. If DoD security certification is an unwaiverable requirement, then the only choice is a less-than-current version of one of the big-company products, like Oracle.

5.6 Distribution and Size of Data

Distribution and size capabilities are not specific to a DBMS architecture. There are significant capabilities in some products within each category and others that have limited capability.

The TRAC2ES project for US Transportation Command (see Appendix E) is an information system to plan and monitor world-wide movement and care of military medical patients. Data about the patient, his condition, and care are loaded into the database by the local medical treatment facility. Information about the available beds and care providers in hospitals world-wide is kept up-to-date by the local hospitals. Information about all military aircraft missions and medical facilities on each aircraft is available to the Global Patient Movement Requirements Center and the Theater Patient Movement Requirements Centers. TRAC2ES analyzes this complex and highly distributed data and plans missions to provide safe, speedy, and cost-effective transportation of patients to the appropriate hospital for care. The system will recommend changes if problems, such as airport closing, missed flights, and changes to hospital bed availability, occur before the patient arrives at his destination.

TRAC2ES has a major requirement for access to evacuation resources and hospital bed data from many locations. The developers of TRAC2ES chose to implement a network of Versant DBMSs. Like TRAC2ES, many DoD applications have the requirements for wide distribution of database access over a network and the ability to cope with data sources that are not always available.

Some object DBMSs have had restrictions on the size of the database primarily because of the limitation of the number of unique object identifiers that can be handled by the DBMS. The major products have addressed that issue in the recent releases by either allowing distribution of data between multiple database instances or allowing the user to specify the length of the object identifier.

5.7 Final Evaluation

A final selection of a commercial persistent storage mechanism from the few candidate products should be based on:

- Examination of other applications that use the candidate products
- Discussions with developers who are experienced with the products
- Discussions with sales and technical representatives
- Examination of available features for use in the specific application, possibly including a small prototype. The prototype may demonstrate to a limited degree the feature interaction and performance of the full system.

6 CONCLUSION

This document presents a summary of the major issues that a developer of object-oriented applications should consider when deciding on a persistent data support environment. Object-oriented technology is growing very fast and promises to greatly improve virtually all aspects of software engineering. The object DBMS market is small but growing in relation to the relational DBMS market. There are a number of financially sound companies that are providing evolving and dependable support to persistent objects.

The ODMG and the OMG standards bodies are coordinating their efforts in a way that will accommodate products that have adopted the SQL standards as well as those that have moved to the ODMG standards. In addition, the object DBMSs are positioned well to provide Persistent State Service (PSS) support in a CORBA environment.

The use of a language-specific database interface vs. an SQL or OQL interface is an early critical decision in the early development of a system. The language specific interfaces are portable across languages and ODMG-compliant products. They provide a single environment for the developer for the application logic and the database. Object-oriented DBMSs and Object-relational mapping DBMSs always provide at least one programming language interface to the database and most also provide an OQL or SQL3 query language interface. Object-relational DBMSs provide at least a variant of SQL and most also provide at least one programming language database interface. Relational DBMSs with object extensions provide only a variant of SQL with specific extensions that support the object classes supported by the add-on module.

The four major types of DBMS architectures should be examined to determine which one is the most appropriate to support an organization's object requirements. They provide different capabilities and are targeted to different requirements. The presence of complex data and complex query patterns are major indicators that the impedance mismatch between the object-oriented programming environment and the relational DBMS may cause problems for an object-oriented application. In this case the developer should estimate the level of effort required to maintain the mapping code between an object-oriented application and a relational database model. A high-maintenance interface is a good indication for the potential of an object DBMS in the target environment instead of a relational DBMS.

The developer should evaluate the features of the object DBMS products and determine the critical requirements of the environment. The developer should evaluate the impact on development and maintenance of the additional mapping code to compensate for impedance mismatch. Examination of the list of requirements using the DBMS Needs Assessment for Objects [2] and other object DBMS documentation should be used to narrow the search to a few candidate products. A final decision should be based on examination of other applications that use the candidate products, discussing issues with experienced developers and sales and technical representatives, and testing key features for use in the specific application.

A. OBJECT STANDARDS BODIES

There are three standards bodies that directly affect the development of databases that store object data. This chapter introduces each of them. Appendix B and C discuss their standards and provide code examples using their standards respectively.

A.1. Object Management Group

The Object Management Group (OMG) is a non-profit corporation that was founded in May 1989 by eight companies: 3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems and Unisys Corporation. OMG now includes over 800 members representing all the major hardware and software vendors as well as system integrators and many commercial and government organizations.

The OMG is dedicated to creating and popularizing object-oriented standards for application integration and the development of distributed applications. OMG's goal is to develop commercially viable and vendor independent specifications for the software industry in order to create a component-based software marketplace. The organization's charter includes the establishment of industry guidelines and detailed object management specifications to provide a common framework for application development.

OMG is developing the Object Management Architecture through its worldwide standard specifications: Common Object Request Broker/ Internet Inter-ORB Protocol (CORBA/IIOP), Object Services, Internet Facilities and Domain Interface specifications (for more information see http://www.omg.org). OMG headquarters is in Framingham, Massachusetts with international marketing partners in the UK, Germany, Japan, India and Australia.

OMG's series of specifications detail the necessary standard interfaces for Distributed Object Computing. These interface standards will make it possible to develop a heterogeneous computing environment across all major hardware platforms and operating systems. The Internet protocol Internet Inter-ORB Protocol (IIOP) is being used as the infrastructure for technology companies like Netscape, Oracle, Sun, IBM and hundreds of others. These specifications are used worldwide to develop and deploy distributed applications for manufacturing, finance, telecommunications, electronic commerce, real-time systems and health care.

For further information contact:

Object Management Group
Framingham Corporate Center
492 Old Connecticut Path
Framingham, MA 01701
Phone 508-820-4300
WEB: www.omg.org

A.2. Object Database Management Group

The Object Database Management Group (ODMG) is a committee representing almost all of the object-database vendors. It was formed in 1991 and their first standard ODMG-93 was released in August 1993. Their latest version, ODMG 2.0, was released in March 1997. Rick Cattell of JavaSoft chairs the committee. Jeff Eastman, of Windward Solutions, is the vice-chair and Douglas Barry, of Barry and Associates, is the executive director. Voting members are from O2 Technology, POET Software, UniSQL, IBEX, Object Design, GemStone Systems, Versant Object Technology, and Objectivity. Reviewing members are from Lockheed Martin, MATISSE Software, VMARK Software, MICRAM Object Technology, MITRE, Electronic Data Systems, Persistence Software, NEC, Fujitsu Open Systems Solutions, Microsoft, ONTOS, CERN, Andersen Consulting, Sybase, Unidata, and Hitachi.

The ODMG 2.0 standard is the *de facto* standard for object DBMSs. Its stated purpose is to ensure the portability of applications across different object DBMSs, protecting the user's software investment while reducing reliance upon a single vendor, and encouraging competitive feature development. The standard is at the intersection of three existing standards domains: databases (SQL), objects (OMG) and object programming languages (C++, Smalltalk, and Java). Rather than defining a completely new standard, the ODMG standard builds upon the existing American National Standards Institute (ANSI) programming language standards and ANSI SQL-92. It also defines a framework for application portability between object DBMSs. The functional components of the ODMG standard include an Object Model, an Object Definition Language, an Object Query Language, and Language Bindings to C++, Smalltalk and Java.

For further information contact:

Object Database Management Group
14041 Burnhaven Drive, Suite 105
Burnsville, MN 55337
Phone: 612-953-7250
WEB: www.odmg.org

A.3. ANSI X3H2

The American National Standards Institute (ANSI) X3H2 standards committee developed the SQL-92 standard and is in the process of developing the SQL3 standards. H2 is a committee under ANSI National Committee for Information Technology Standards (NCITS, pronounced "insights", formerly ANSI X3). This committee was first formed in 1978 to develop the Network Database Language (NDL) for the network database model. In 1982 it was given the responsibility to develop SQL for relational databases. The current SQL3 project is an effort to extend SQL-92 with an object capability. ANSI is the US representative on the International Organization for Standardization (ISO) where the JTC1 committee handles SQL standards.

For further information contact:

NCITS, Secretariat, Information Technology Industry Council (ITI)
1250 I Street, NW, Suite 200
Washington, DC 20005-3922
WEB: www.ncits.org
or www.jcc.com/sql_stnd.html for SQL standards

B. OBJECT MODELS AND DATA LANGUAGES

This section discusses the object models and languages of the OMG, ODMG, and the ANSI X3H2 groups that are related to databases. Although SQL is not a computationally complete language, it does include a Data Definition Language (DDL), a Data Manipulation Language (DML), and a Data Query Language (DQL) with embedded capability. The ODMG provides this type of capability in three specifications. The ODMG Object Query Language (OQL) provides only a read-only query capability. The ODMG Object Definition Language (ODL) is separate and provides mechanisms for object definition. ODMG Object Manipulation Language (OML) is supported only through the language bindings to C++, Smalltalk and Java. The OMG Persistent State Service (PSS) is closely related to the ODMG ODL and the DDL portion of SQL. The OMG Query Service (QS) specification specifies an environment where ODMG OQL and the query statements of SQL can be executed and values returned.

B.1. ODMG and OMG Object Model

The ODMG Object Model is the basis for the entire ODMG standard. The ODMG Object Model is a strict superset of the OMG Object Model. ODMG extends the OMG model with capabilities like persistence, extents (the set of all instances of a class), keys, relationships between objects, exceptions, queries, and transactions to support database functionality. The OMG assumes that there will be an exception handling routine defined outside the core model. In one place the two models have a different name for the same concept. The OMG "non-object" is the same as the ODMG "literal," which is an elemental data type (e.g., character, integer) that does not necessarily have an associated object identifier.

The key concepts of the ODMG object model include:

- Attribute and relationship object properties
- Object operations (behavior) and exceptions
- Multiple inheritance
- Extents (the set of all instances of a class) and candidate keys
- Object naming, lifetime and identity
- Atomic, structured and collection literals

- List, set, bag and array collection classes
- Concurrency control and object locking
- Database operations

B.2. ODMG Object Definition Language

All DBMSs provide a DDL that enables the user to define the schema. The ODMG ODL is a database schema definition language. It provides a portable definition of the database schema, operations, and state of the database for all object-database products. It is a strict superset of the OMG Interface Definition Language (IDL) with support for the extensions of the ODMG data model over the OMG data model.

ODL creates a layer of abstraction such that an ODL-generated schema is independent of both the programming language and the particular ODMG-compliant DBMS. Accordingly, ODL considers only object type definitions, including method signatures but not actual implementation of methods. An ODL-generated schema can be freely moved between compliant DBMSs, different language implementations (e.g., Java, C++, and Smalltalk), or even translated into other DDL, such as those proposed by SQL3.

B.3. OMG Persistent State Service

The OMG has an approved Persistent State Service (PSS)[6] specification (formerly Persistent Object Service) that the major object-database vendors are supporting. A request for proposal (RFP) has been released for version 2.0 of the Persistent State Service with revised submissions due in May 1998. Four proposals[7] were submitted in November 1997 by: Fujitsu, EDS, Secant Technologies, and a group of eleven companies led by SunSoft. The new PSS will be an extension of the previous specification that will emphasize the fact that an object may store its state in a PSS. The proposal from the group of eleven companies led by SunSoft is heavily influenced by the ODMG specification. The high-level mechanisms in the new PSS specification will be essentially the same as the previous persistent object specification. This section discusses the approved specification with the understanding that many of the terms will soon change and many of the functions will be extended but the general architecture will illustrate the CORBA environment for PSS.

The goal of this service is to provide common interfaces to the mechanisms used for retaining and managing the persistent state of objects. The OMG IDL is used to specify the interfaces to these mechanisms within the CORBA environment. It also defines the conventions by which objects can work together using the PSS. To support the persistent state of objects the PSS works with other CORBA services such as naming, relationships, query, transactions, life-cycle, security, and object-by-value.

The major components of the currently approved PSS (see Figure 6) are as follows:

- Persistent Identifier (PID): This describes the location of an object's persistent data in some Datastore and generates a string identifier for that data. Any new PSS will have some mechanism for locating a datastore.
- Persistent Object (PO): This is an object whose persistence is controlled externally by its clients. Any new PSS will have objects with persistent state.
- Persistent Object Manager (POM): This component provides a uniform interface for the implementation of an object's persistence operations. An object has a single POM to which it routes its high-level persistence operations to achieve plug and play. Any new PSS will have an interoperability mechanism across platforms and PSS implementations.
- Persistent Data Service (PDS): This component provides a uniform interface for any combination of Datastore and Protocol, and coordinates the basic persistence operations for a single object. It translates between the object view and the storage view using one of the protocols. Any new PSS will have a set of standard interfaces to the persistent state of the object.
- Protocol: This component provides one of several ways to get data in and out of an object. The three protocols are Direct Access Protocol, the ODMG-93 Protocol, and the Dynamic Data Object Protocol. Any new PSS will have a set of standard interfaces to the persistent state of the object.
- Datastore: This component provides one of several ways to store an object's data independently of the address space containing the object. Object DBMSs are ideal candidates for datastores because they already use the ODMG-93 protocol, which is a subset of the functionality defined in the ODMG-93 standard.

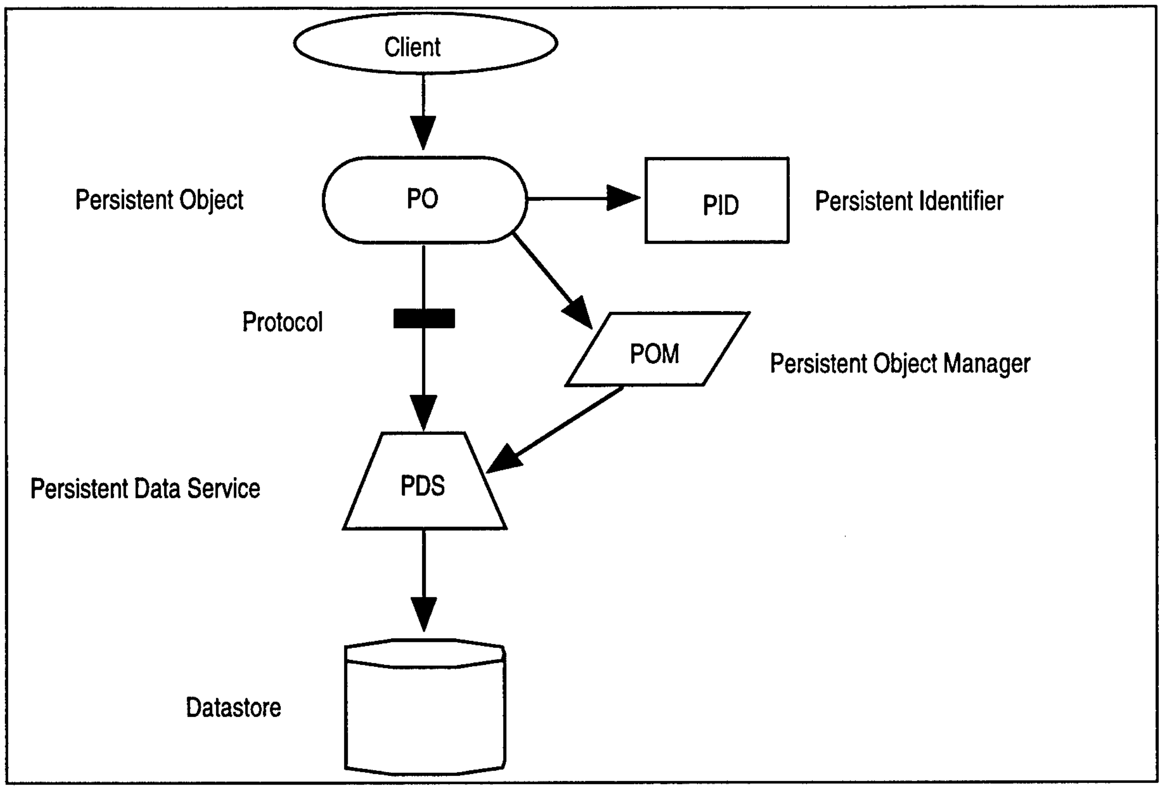


Figure 6. Major PSS Components

CORBA services usually deal with course-grained objects such as files, services, servers, databases, etc. With the implementation of the PSS and query services, the programmer has the option of registering finer-grained objects with the PSS.

B.4. OMG Query Service Specification

Like the PSS, the goal of the Query Service (QS) specification is to provide common interfaces to the mechanisms used for query within the CORBA environment. The QS provides the IDL-specified format for issuing queries to arbitrary collections of objects, including the ability to specify values of attributes, invoke arbitrary operations and invoke services within the CORBA environment, such as the relationship service. It includes the ability to select, insert, update, and delete objects to a Persistent State Service. It supports a potentially nested and federated set of query evaluators. It provides definitions and interfaces for creating and manipulating collections of objects that are the results of queries to native DBMS. It also provides iterators that are used to traverse and manipulate the collections. The current specification allows queries using either the form of SQL-92 or ODMG-93 with words specifying that when new versions of the two standards are available they should replace the older ones. Therefore ODMG version 2.0 is now one of the two standard query languages specified by QS.

Object DBMS that support ODMG 2.0 can be easily modified to be used as a datastore in the CORBA environment because they are already consistent with both the PSS and the QS specification. Several object DBMSs are already marketing their products as being "CORBA-compliant." Using the OMG PSS and QS interface when building client applications also results in minimal modification if one datastore implementation needs to be replaced by another.

B.5. ODMG Object Query Language

Object Query Language (OQL) is an SQL-like declarative language that provides a mechanism for efficient querying of database objects, including high-level primitives for object sets and structures. OQL provides almost all of the read-only query capability of SQL-92. Most SQL select statements that run on relational DBMS tables work with the same syntax and semantics on ODMG collection classes.

OQL also includes object extensions to support object identity, complex objects, path expressions, operation invocation and inheritance. OQL may invoke operations in ODMG language bindings, and OQL may be embedded within a programming language for which an ODMG binding is defined; currently Java, C++, and Smalltalk. OQL maintains object integrity by invoking an object's defined methods, rather than using update operators specified by an OQL query. The ability to invoke methods contained in the objects gives OQL the ability to query and update values of attributes and insert and delete objects without violating encapsulation.

B.6. SQL3 Query vs. OQL

SQL3 is under development by the ANSI X3H2 committee with a goal to extend SQL-92 to include object capability. In an effort to ensure interoperability between OQL and the SQL3 proposal, the ODMG has created a strong working relationship with the ANSI X3H2 committee. This effort has been ongoing for at least two years. Some of the compromises between the two groups have been captured in the new ODMG 2.0 standard such as the capability of handling null values.

There are differences in syntax and semantics between an OQL query to an object DBMS and an SQL query to a relational DBMS that must be accommodated in the unification of the SQL3 and OQL models. Key functionality of SQL environments, such as tables, domain values, constraints, triggers, views, and virtual transient tables are all capabilities that must be supported by the new object DBMS environments to use the future SQL3/OQL-merged query language. These are not trivial issues but there is an emerging consensus on a solution.

In a unified SQL3/OQL environment, relational DBMS products must support an object identifier that uniquely identifies an object, independent of the values of its attributes/columns. These values must be stored in a relational DBMS in a way that they either identify the row or reference other rows of other objects.

Probably the most fundamental difference in the OQL and SQL models is in the values that are returned. The SQL query returns a list of values; normally character values that are interpreted by the receiving client based upon the structure of the query. The result can be viewed as a virtual table but the client does not process tables; only values. Many database interfaces supplied by programming environments provide mechanisms for formatting the returned values into records, which are defined by the programmer for each query. In this case the result is interpreted as a list of records.

In OQL the result is a bag (unsequenced set with possible duplicate values) of a structure defined in the query. The structure is either a database object or an application-defined structure. In some cases, the structure may only contain object identifiers to objects. One approach that has been proposed for this inconsistency between the two query languages is to rename the ODMG select statement as a select_object statement. The select_object statement would return a bag of structures and the select statement would return a list of values.

In the near future, unification of the SQL and OQL may allow an SQL query to read data from an object DBMS and OQL to read relational DBMS data, but creating and updating data may have challenges that will not be solved immediately. This is because there is a fundamental conflict between the object-oriented and relational paradigms. The SQL approach of creating, updating, and querying all data by an external program violates basic object-oriented encapsulation principles. In object-oriented systems data is accessible outside the object only via its public interfaces.

As stated in Section B.2, OQL, ODL and OML language bindings are separate ODMG specifications, and SQL includes all three specifications. ODMG requires that all updates to data in the database be accomplished exclusively via invocation of methods using OQL or method invocation using one of the object-oriented programming language bindings. They provide a mechanism for specification of the public object interface to the outside world via ODL (just as IDL does in the OMG/CORBA), but the native programming language is always used for method development.

To access objects from an object DBMS using OQL or a language binding the object must have a public method that returns the result requested. SQL only requires that a named column exists in a table for permission to access the data. There is no concept of public and private data other than that established by security mechanisms in each relational DBMS. In object systems some of the "values" are calculated based on code contained in the corresponding accessor method rather than being stored in an attribute. An example is the age of a person. The "get_age" method may derive a value from system_date and a "date_of_birth" attribute rather than an "age" attribute. Any calculation of this type in a relational system must be embedded in every query or in a stored procedure.

The object-oriented paradigm does not allow the attributes of an object to be updated by code outside the object. Values may be indirectly updated by invoking public method interfaces that pass values to an object that are used to update internal values. Calls to these public interfaces cause the object to update its internal attributes in ways that are unknown to the calling code. This architecture allows changes to occur to the internal data structure that will not affect the code outside the object. These points illustrate why ODMG made such a distinction between the roles of query, definition, and manipulation of data.

This ability to hide the implementation of an object and expose only the public interface is often overlooked by simple examples similar to the ones used in this paper. By default, object DBMSs create an accessor method for each public attribute if one is not defined by the programmer. Simple examples with only public attributes do not illustrate the ability to hide or tailor access to private attributes.

Allowing an SQL statement, or a program, using an ODBC interface, to directly update values of an object in an object DBMS, violates the object-oriented paradigm. It does not use the method interface that was designed for update of values in the object.

B.7. Language Bindings

The ODMG C++, Smalltalk and Java bindings define OMLs that extend the native languages to support retrieval and update of persistent objects in object DBMSs. The bindings also include support for OQL, ODL, navigation and transactions.

Because each language has its own ODL and OML, developers can work inside a single language environment without separate programming and database languages. Although language-independent OQL and ODL are supported by most object DBMSs today, reliance on a single programming language for code and database interaction has been one of the most fundamental differences between the object DBMSs and the relational DBMSs. The argument for language interfaces has been that it provides one object-oriented language for development of both applications and databases. The data structure is the same in the database and the application. The database can be developed and maintained by the application developers without learning SQL or any other database-specific language.

The interoperability and interchangeability of object DBMS products have been improving. ODMG 2.0 [4] includes a specification language, called the object interchange format, for exchanging objects between object DMBSs. In recent years, the major object DBMS products have allowed data written using one language interface to be read using any of the other supported language interfaces. Even if an object DBMS was replaced with another object DBMS the code does not need to be changed significantly (except for product-specific extensions).

B.7.1. C++ Language Binding

The ODMG C++ binding provides ODL and OML translation to the C++ environment. OQL is a distinguished subset of OML. The C++ binding allows persistence-capable classes to be created by inheritance from a database-specific C++ class library. These C++ classes provide facilities for object creation, naming, manipulation, deletion, transactions, and other database operations. The binding also includes recent ANSI C++ enhancements, such as support for Standard Template Library (STL) algorithms for sequential access to members of a collection and exception handling.

The similarity between normal C++ language mechanisms, C++ ODL, and C++ OML gives the programmer a single language environment instead of separate programming and database languages. This unified environment has a single type system where individual instances of the types can be either persistent or transient.

B.7.2. Smalltalk Language Binding

The ODMG Smalltalk binding provides the ability to store, retrieve and modify persistent objects in Smalltalk. The binding includes a mechanism to invoke ODL, OML, and procedures for transactions and operations on databases. Smalltalk objects are made persistent by reachability. This means that an object becomes persistent when it is referenced by another persistent object in the database, and it is garbage-collected when it is no longer reachable.

The Smalltalk binding directly maps all of the classes defined in the ODMG object model to Smalltalk classes. Object Model collection classes and operations are mapped wherever possible to standard Smalltalk collection classes and methods. Relationships, transactions and database operations are also mapped to Smalltalk constructs.

B.7.3. Java Language Binding

The ODMG language binding for Java is new in Release 2.0 of the ODMG standard. The binding includes a mechanism to invoke ODL, OML, and procedures for transactions and operations on databases. It adheres to the same principles as the Smalltalk and C++ bindings. The binding uses established Java language practice and style so as to be natural to the Java environment and programmers. Instances of existing classes can be made persistent without changes to source code. As in the Smalltalk binding, persistence is by reachability: once a transaction is committed, any objects that can be reached from root persistent objects in the database are automatically made persistent in the database. The ODMG binding to Java adds classes and other constructs to the Java environment to support the ODMG object model, including collections, transactions and databases.

C. EXAMPLE

This chapter shows representative code examples using the language standards related to relational and object DBMSs. The developer should review these examples to understand the basic similarities and differences in the choice of database interface languages.

This chapter uses a baseball league database as an example. It keeps statistics of teams and players in a series of games over time. It records data on ticket sales and promotions. Object and relational models of this database are included for comparison.

C.1. Partial Object Model for Baseball

Figure 7 is a partial class diagram of a baseball domain. It involves players that play for a team for a period of time, and games between home and visitor teams that are played at stadiums on specific dates and times. Players play specific positions and accumulate statistics in those positions and at bat. Tickets for specific seats and games are bought by fans and some fans own a list of tickets. Sales Promotions are associated with Games.

The diagram is not a complete object model but it is included for comparison with a similar relational model. It shows only a few methods associated with the classes to illustrate how they are specified on this type of diagram. The behavior of an object is specified by the methods associated with the object. Other diagrams would also be required to complete an object model that would specify the attributes, associations, and behavior of the domain (e.g., object interaction diagram, state diagram, activity diagrams).

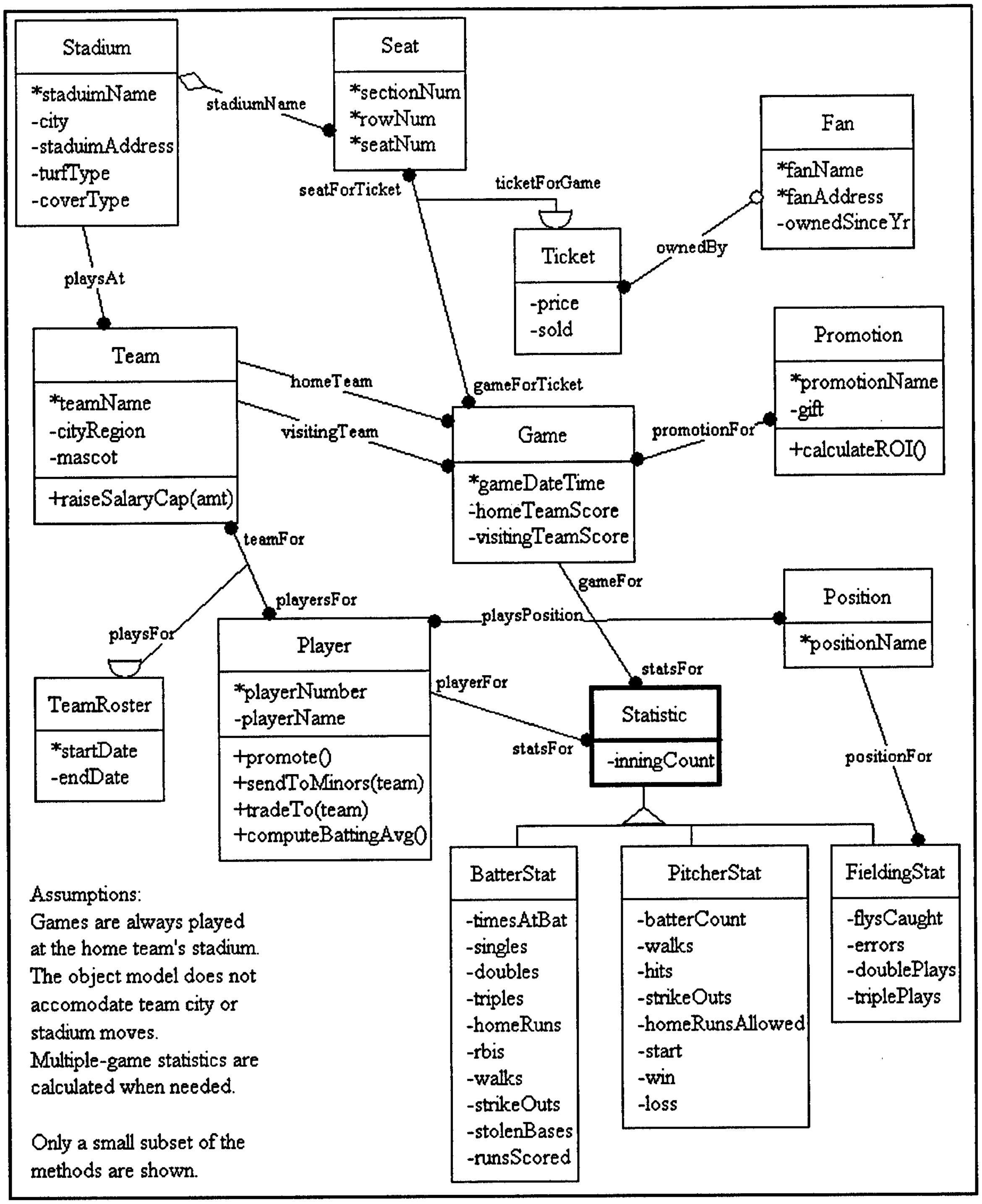


Figure 7. Baseball Class Diagram

The notation used is Rumbaugh's Object Modeling Technique [8],which is incorporated into Unified Modeling Language (UML). It represents classes (object types) within a box, with the name of the object in the top section, and the attributes in the next section, and methods in the lower section. Associations between the objects are identified by lines between the objects with solid circles at an end indicating a multiple association. For example, a Game may have zero or many Promotions and a Promotion may have zero or many Games. A hollow circle indicates an optional association. For example, a Ticket may or may not be ownedBy a Fan. The lines labeled with a triangle indicate a generalization association, which is a superclass to subclass relationship. For example, a PitcherStat is a subclass of Statistic. Each subclass of Statistic inherits the association playerFor and gameFor and the attribute inningCount. Lines labeled with a half circle are association classes, which are classes that contain attributes about the association between two other objects. For example, the TeamRoster contains information about the association between a Team and a Player. Attributes preceded by an asterisk are potential key values. Lines labeled with a diamond are aggregation association where the opposite end is a component of the diamond end. For example, seat is a component of a Stadium.

It is important to examine the complexity of the diagram, especially for many-to-many relationships. In this model there are many-to-many relationships between Team and Player, between Seat and Game, between Player and Position, and between Game and Promotion. The first two have association classes for the many-to-many relationships. For example the Ticket captures the price and a "sold" indicator about the relationship between a Seat and a Game. There is one and only one Ticket for each Seat and Game combination.

The example in Figure 7 is not very complex as it is depicted even though there are four many-to-many associations and the Game and Ticket classes have complex key structures. If this schema were expanded for use by a market analysis program, the schema may become very complex. For example, complex data may be required for ticket sales with regard to the weather, team win-loss record, big-name players, key players with specific ethnic backgrounds, how often someone gets into an argument with an umpire, visiting teams from specific parts of the country, or inter-league play. Including video and audio recordings in the database and referencing sections of them to a recorded database event would add considerable complexity. Any of these factors may cause the model to contain more complex subclass relationships and other associations between classes.

C.2. Relational Model for Baseball

Figure 8 represents the schema of a baseball domain in a relational model. It is in Entity-Relationship notation. The dotted lines indicate an optional association. The table names are above the box, with the primary key fields in the upper section, and the remaining fields in the lower section. To implement this baseball database in a relational model the remaining two many-to-many relationships must be captured by an association table including only the keys to the two associated tables (PlayerPosition and GamePromotion). The "FK" in parenthesis after the row name indicates that it is a foreign key and is used to join that table with the primary key of another table.

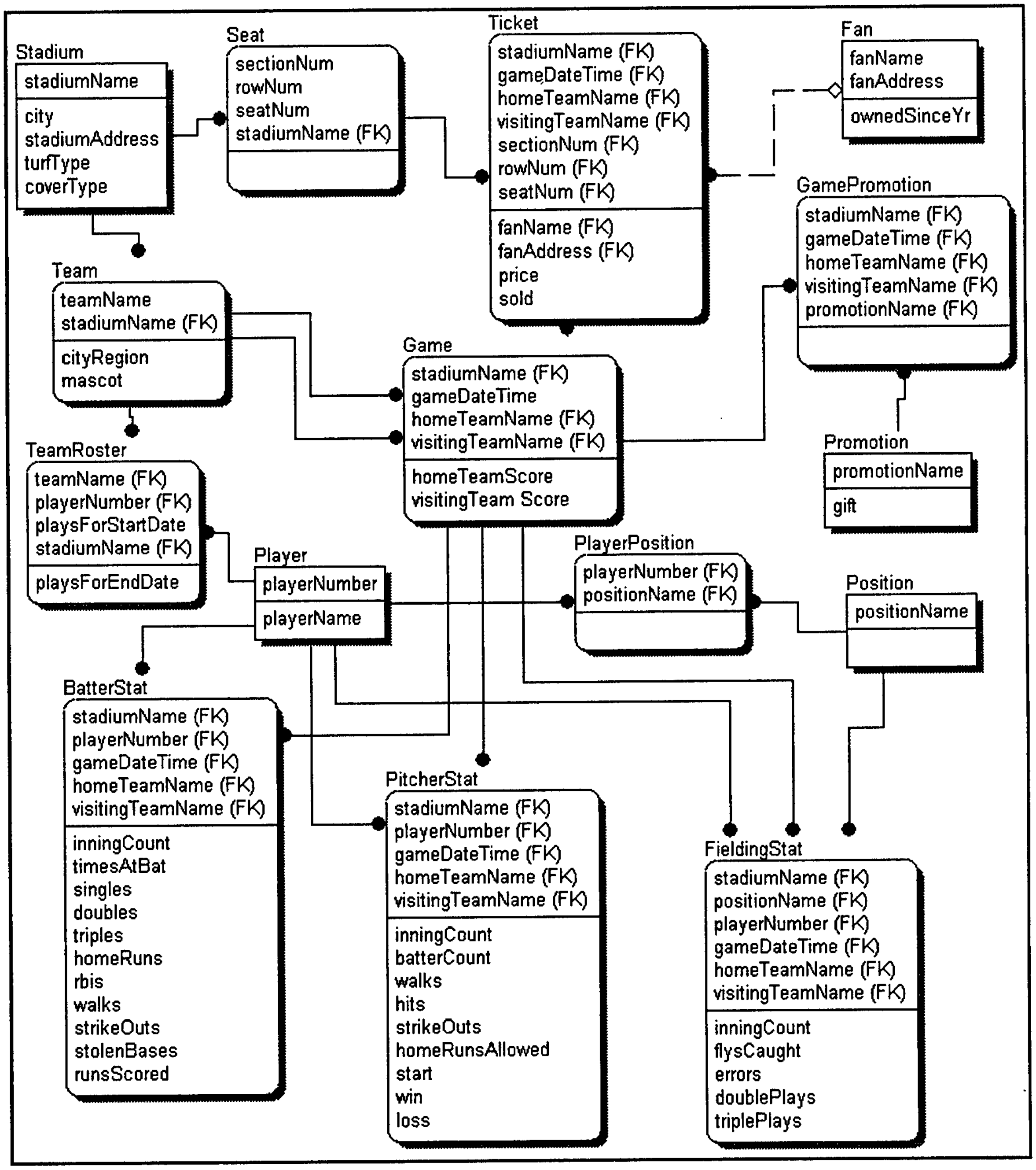


Figure 8. Baseball Relational Model

C.3. Example ODMG ODL

Figure 9 is an example of ODMG ODL that creates the Game and Promotion classes in Figure 7.

```
class Game
(           extent Games)
               // defines an extent of the class Game in the baseball database
{
            attribute   timestamp gameDateTime;
            attribute int        homeTeamScore;
            attribute int        visitingTeamScore;
            relationship set<Promotion> promotionFor
                        inverse Promotion::promotionFor;
            relationship set<Ticket> ticketForGame
                        inverse Ticket::gameForTicket;
            relationship Team homeTeam
                        inverse Team::homeTeam;
            relationship Team visitingTeam
                        inverse Team::visitingTeam;
};
class Promotion
(           extent Promotions)
            // defines an extent of the class Promotion in the baseball database
{
            attribute string promotionName;
            attribute string gift;
            relationship set<Game>  promotionFor
                        inverse Game::promotionFor;
            float calculateROI();
};
```

Figure 9. ODMG ODL Example

In this example, there is a bi-directional relationship between Promotion and Game that has a name of "promotionFor" in both directions. (Each end of a relationship may have different names, such as the relationship between Team and TeamRoster.) The ODL for the relationship attribute specifies the name of the class for the other end of the relationship and the name of the association in that class capturing the inverse relationship.

An extent of a class is a set of all of the instances of the class. If an extent is defined as in the two classes of this example, it is automatically maintained by the object DBMS as objects are created or deleted. Definition of an extent is optional.

Specification of operations is almost identical to C++ header files. The Promotion class has one operation specified in Figure 9 that calculates the return on investment for a Promotion to see if it is affecting ticket sales. The code that executes the calculateROI method would be specified in C++, or Java, or Smalltalk. The ODL operation signatures are specified by the return type, the name of the operation, a parameter list with types, and the name of any exceptions. The specific language bindings are used to specify the behavior of the operations.

C.4. Example C++ ODL

Figure 10 is an example of C++ ODL that creates the Game and Promotion object. The syntax is similar to the ODMG ODL. (We could have shown these examples in Java or Smalltalk.) The example in Figure 9 implements the same two objects using ODMG ODL as in Figure 10 using the C++ version of ODL.

```
extern const char _promotionGame[];
extern const char _gamePromotion[];
extern ...
class Game : public d_Object {
public:
// properties:
            d_Timestamp                                 gameDateTime;
            unsigned short                              homeTeamScore;
            unsigned short                              visitingTeamScore;
            d_Rel_Set<Promotion, _gamePromotion >       promotionFor;
            d_Rel_Set<Ticket, _tickets>                 ticketForGame;
            d_Rel_Ref<Team, _homeTeams>                 homeTeam;
            d_Rel_Ref<Team, _visitingTeams>             visitingTeam;
            d_Extent                                    Games(baseball)
                      // defines an extent of the class Game in the baseball database
// operations         none shown for this class
private:
            ...
};
class Promotion : public d_Object {
public:
// properties:
            d_String                              promotionName;
            d_String                              gift;
            d_Rel_Set<Game, _promotionGame>       promotionFor;
            d_Extent                              Promotions(baseball)
                      // defines an extent of the class Promotion in the baseball database
// operations
            float                                 calculateROI ();
private:
            ...
};
const _promotionGame[] = "promotionFor";
const _gamePromotion[]"promotionFor";
```

Figure 10. C++ ODL Example

For a comparison of the creation of the Game and Promotion tables in SQL see Figure 32 and Figure 33. The ODL is very much like the C++ environment of the program. The Java and Smalltalk ODL would look very much like their respective programming languages. The SQL code to create the two tables does not resemble any object-oriented programming language and it is focused on defining relational constructs such as primary and foreign keys.

C.5. Example C++ OML

C.5.1. New Object Creation

In C++ OML, new objects of a type are created with the new operator with an extra parameter to specify a transient object or the name of the database for a persistent object.

The example in Figure 11 illustrates the use of smart pointers or reference variables. Every persistence capable class has a corresponding reference class that refers to the class. Game1 references Game, prom2, temp_prom3, and temp_prom4 reference Promotion.

```
d_Database *baseballdb;              // declare a database of the name baseballdb
                                     // assume proper initialization of database

d_ref<Game> game1=new(baseballdb, "Game") Game;
            // create a new Game object in the baseballdb database referenced by game1
d_ref<Promotion> prom2=new(game1, "Promotion") Promotion;
            // create a new Promotion object in baseballdb near the game1 object
            // referenced by prom2
d_ref<Promotion> temp_prom3=
               new(d_Database::transient_memory, "Promotion") Promotion;
            // create a new transient Promotion object referenced by prom3
d_ref<Promotion> temp_prom4=
               new Promotion;
            // create a new transient Promotion object referenced by prom4
            // this is an alternate form of the statement above that created temp_prom3
```

Figure 11. C++ OML Creation Example

C.5.2. Object Deletion

The references to the deleted objects are made in memory immediately and the changes are made in the database upon transaction commit as shown in Figure 12.

```
game1.delete_object();
            // deletes the persistent object
            // game1 will have an undefined value
```

Figure 12. C++ OML Deletion Example

C.5.3. Open a Database

C++ OML allows a simple means of declaring a database variable and opening the database associated with that variable as shown in Figure 13.

```
static d_Database *baseballdb;       // declare a database variable of the name baseballdb
...
main()
{
baseballdb.open("databaseName");  // open the database with the name of databaseName
                                  // baseballdb will refer to this specific database
...                               // main body code
baseballdb.close();               // close the database
}
```

Figure 13. C++ OML DB Open Example

C.5.4. Name an Object

C++ OML uses three simple methods for naming objects within a database. Given the baseball schema as defined in Figure 7, the three methods are shown in Figure 14:

```
d_Ref <Game> game1, game2;
// not shown: set game1 to a specific Game object

baseball.set_object_name(game1,"OpeningGame");
            // sets the name of the object referenced by game1 to "OpeningGame"
baseball.rename_object("OpeningGame","ClosingGame");
            // changes the name of an object from "OpeningGame" to "ClosingGame"
game2 = baseball.lookup_object("ClosingGame");
            // retrieves the object named "ClosingGame" and assigns the object to game2
```

Figure 14. C++ OML Name Example

C.5.5. Transactions and Locks

Transactions and locks are necessary to control updates to the database. In some cases changes to the database must be made in several places to be consistent. For example, if an error occurs during the processing of annual salary increases for a company that prevents all employee records from being updated, all updates should be removed to ensure that all employee records are updated once and only once. Other database users should not be able to see any changes until all of them have completed successfully. The C++ OML for transactions is shown in Figure 15:

```
d_Transacton firstTwoObj;     // declare a transaction variable
firstTwoObj.begin();          // begin a transaction
...                           // not shown : actions taken during transaction
firstTwoObj.commit();         // end the transaction and commit changes to the database
OR
firstTwoObj.checkpoint();     // commit changes to the database made since the
                              // last checkpoint. Leave the transaction and its locks
                              // in place.
OR
firstTwoObj.abort             // discard changes and release locks
```

Figure 15. C++ OML Transaction Example

Transaction objects are transient variables and are subject to normal rules about scope. Long transactions (a transaction that lasts longer than the process that created it) are supported in many object DBMSs but are not included in the current ODMG standard.

The ODMG model uses pessimistic concurrency control as a default but many vendors support other control policies. A program may use either implicit or explicit locking. In explicit locking a programmer will use the lock and try_lock operations. In implicit locking read locks are obtained when the objects are accessed in the database. Write locks are obtained when the referenced data is modified in the client workspace. The programmer may set a C++ OML preprocessor switch to automatically detect when persistent objects are to be modified. The preprocessor switch default setting is off. If the switch is not set, the programmer must notify the object DBMS runtime process that a persistent object will be modified with the statement in Figure 16:

```
game1.mark_modified();
            // notifies the object DBMS runtime process that a Game object
            // is going to be modified
```

Figure 16. C++ Modified Example

C.5.6. Object Modification

A programmer modifies a persistent object associated with a reference variable in the same way that any C++ object is modified. The changes are made only in memory until a transaction commit, when the changes are propagated to the database. If the programmer sets a C++ OML preprocessor switch to automatically detect when persistent objects are to be modified there is virtually no difference in the programmer's treatment of persistent and transient objects other than the declaration of the variables. If the switch is not set then the programmer must use the mark_modified method described in Section C.5.5.

Referential integrity of bi-directional relationships is maintained by the object DBMS. If one of the objects is deleted, inverse relationships are altered to remove the reference to the deleted object. The same is true when a new relationship is added to an object. In Figure 17, the insert_element method is invoked on the promotionFor set to add a reference to a specific game to the relationship. The object DBMS would keep the inverse attribute synchronized.

```
prom2.promotionFor.insert_element(& game1);
            // inserts the game1 object into the prom2.promotionFor attribute
            // object DBMS inserts prom2 into the game1.promotionFor attribute
```

Figure 17. Referential Integrity Example

C.6. Example ODMG OQL

OQL may be used to query a "named objects" in a database that may be an atom, a structure, a collection, or a literal. The OQL query is a function that returns a single object that is also an atom, a structure, a collection, or a literal. Most often queries are executed on collections. Extents and sets of references to associated objects are very important in this respect. An example of a simple query from the baseball database is shown in Figure 18

```
select t.cityRegion
from Teams t          // Teams is the extent for Team
where t.teamName = "Orioles"

OR

select calculateROI()
from Promotion p
where p.promotionName = "bat"
```

Figure 18. OQL Single Value Example

The result of the first query is a literal bag of strings with the city or region name of all of the baseball teams named "Orioles." In this case it is a single string of "Baltimore."

The result of the second query is a bag containing one floating point number that is an indicator of the return on investment for the bat promotion. How it is calculated is not important here. The details are hidden within the implementation of the Promotion object.

Assuming that the Orioles have only been based in one city the query shown in Figure 19 will return a single object of type Team.

```
select t
from Teams t            // Teams is the extent for Team
where t.teamName = "Orioles"
```

Figure 19. OQL Single Object Example

Arbitrary structures may be defined by nesting statements and using the C++-like "struct" construct. Figure 20 shows nested statements and also illustrates path traversal from one object to another.

```
select struct(name: p.playerName,
              pos: (select y from p.playsPosition y))
from Players p          // Players is the extent for Player
where p.playerName = "Cal Ripken"
```

Figure 20. OQL Structure Example

This statement returns a bag of structures with player names and a set of Position objects that the player has played in the database (identified by name and pos). In this specific database the returned value is a set containing one structure. The structure contains the name "Cal Ripken" and a set of two Position objects that have a positionName of "Short Stop" and "Third Base."

Path traversal is a powerful mechanism in object DBMSs. It can simplify queries dramatically. Figure 21 contains an example of a query to retrieve a bag containing all the pitchers with teams that are based in domed stadiums. It involves five classes.

```
select distinct p
from Players p
where p.playsPosition.positionName = "Pitcher"
and p.playsFor.teamFor.playsAt.coverType = "Dome"
```

Figure 21. OQL Path Traversal Example

Where a relation is not defined a path traversal is not possible. Yet if the programmer knows that there is a relationship between two classes, a join may be used that is similar to the join in a relational DBMS. Figure 22 contains an example of a join on the extents of Player and Fan, that returns a bag of strings containing the names of players who are listed in the database as ticket holders.

```
select distinct p.playerName
from Players p, Fans f
where p.playerName = f.fanName
```

Figure 22. OQL JoinExample

Similar constructions are available in OQL, as in SQL, to group and manipulate collections during a query (e.g., count, max, sum, exists, order by, group by).

Optional OQL syntax is allowed in a few places. A "->" is used instead of the period for method invocation, attributes, and path traversal. The "alias" after the name of the collection in the from clause may be omitted or it may take an alternate form as in Figure 23.

```
select distinct Players->playerName
from Players, Fans
where Players->playerName = Fans->fanName

OR

select distinct p.playerName
from p in Players, f in Fans
where p.playerName = f.fanName
```

Figure 23. OQL Alternate Forms

C.7. Example C++ Query

C++ has a highly developed capability to manipulate collections of objects. Any collection that can be referenced in the database via "d_Ref_Any" can be read and iterated over using the same mechanisms as transient collections. One of these methods is "query" to filter a collection based on a predicate that is enclosed within quotes.

In the example in Figure 24, the collection of references to Game, is called homeWins. It is assigned each of the Games in the Collection myTeam.homeTeam that satisfy the predicate within quotes. The cardinality of the resulting collection assigned to homeWins will be the number of wins at the home field. The predicate takes the syntax of the where clause of an OQL statement.

```
d_Bag<d_Ref<Game>> homeWins;
// homeWins is a collection of references to Games
Team myTeam;
// myTeam is a Team object
// not shown: initialize myTeam;

myTeam.homeTeam.query
            (homeWins,
            "this.homeTeamScore > this.visitingTeamScore")
// query all the Games played by myTeam
// assign the variable homeWins the value of the predicate in quotes
// predicate returns the Games that myTeam won at home
```

Figure 24. C++Query Method Example

C++ may also embed OQL statements within C++ code. It uses the methods d_oql_query to define a query and d_oql_execute to execute it The query may also be defined to pass parameters of the form "$1".

The example in Figure 25 executes the query q1, which retrieves the team with teamName = Orioles and assigns the value to the collection myTeams.

```
d_Bag<d_Ref<Team>> myTeams;
d_oql_query q1(" select t from Teams t
                where t.teamName = \"Orioles\" ");
                        // Teams is the extent for Team
d_oql_execute(q1, myTeams)
```

Figure 25. C++ Embedded OQL Example

C.8. Example SQL

This section shows queries that are similar to the previous sections using SQL against a relational version of the baseball database (see Figure 8). The query example in Figure 18 (section C.6) against a relational model using SQL would have a very similar structure. Table names of a relational DBMS are semantically equivalent to the class extents of an object DBMS so the table name is used in place of the object extent. Otherwise the syntax of the two examples is the same. The SQL statement is in Figure 26.

```
select t.cityRegion
from Team t
where t.teamName = "Orioles"
```

Figure 26. SQL Single Value Example

The result is a simple list of strings instead of a bag of strings. In this case, it is a list of one string, "Baltimore."

The query in Figure 20 would not work as written in SQL because it does not allow structures or objects returned as values. A join condition would also be defined in place of the path traversal. The relational form to return similar values is in Figure 27.

```
select p.playerName, q.positionName
from Player p, Position q, PlayerPosition pp
where p.playerName = "Cal Ripken"
and p.playerNumber = pp.playerNumber
and q.positionName = pp.positionName
```

Figure 27. SQL Lists Instead of Structures

The result of this query is a list of values alternating between playerName and playerPosition. In this case it is "Cal Ripken", "Short Stop", "Cal Ripken", "Third Base". The name is repeated because it matched two Positions in the database. *The object-oriented applications, written in C++, Java, or Smalltalk, executing the SQL query would have the responsibility to parse the list of strings in a way that the values are captured in the application.*

The last two lines of the SQL statement in Figure 27 capture the join condition of the three tables. In this very simple example the SQL statement has become bulky even though the two primary tables have a single value as the primary key. The ticket table in Figure 8 has seven values as its primary key and all seven may be required for a join of associated tables. If many tables must be joined the statements can get very awkward.

An example, in the relational baseball database, to retrieve data on all the games that had a 'baseball' as the promotional gift the query would be as shown in Figure 28.

```
select game.*
from Game g, GamePromotion gp, Promotion p
where g.gameDateTime = gp.gameDateTime
        and g.homeTeamName = gp.homeTeamName
        and g.visitingTeamName = gp.visitingTeamName
        and gp.promotionName = p.promotionName
        and p.gift = 'baseball';
```

Figure 28. SQL Game Promotion Query

This query returns a list of values that includes each of the three game attributes for every game that had a baseball as the gift. The object-oriented applications, written in C++, Java, or Smalltalk, must process this list in a way that would create Game objects in the application memory and assign the individual values from the list to the attributes of the objects. This is another example of "impedance mismatch" where the database structure is significantly different than the application object structure.

The query in Figure 19 returned a bag of team objects. Since relational systems will not return objects or structures, a list of all of the attributes of a team is returned. Figure 29 shows a simple query using the "*" feature (for all values) but the result is a list with four strings for each team matching the query. In this case only one row is returned but it could be thousands and the application would have to iterate through the list and assign each value to appropriate variables.

```
select t.*
from Team
where t.teamName = "Orioles"
```

Figure 29. SQL Single Set of Values Example

Since path traversal is not available in a relational DBMS, joins must be used to show relationships between tables. Figure 21 contains an OQL example of a query to retrieve a bag containing all the pitchers with teams that are based in domed stadiums. It involves five classes. The comparison of the following SQL query in Figure 30 with Figure 21 illustrates the relative awkwardness of maintaining code for joins. If one of the join conditions is not accurate then the query may complete but will give an incorrect answer. This example also shows the added overhead of the PlayerPosition table that was required in the relational model in order to eliminate the many-to-many relationship. This query joins five tables but the equivalent query in Figure 21 traversed four classes.

```
select distinct p.*
from Player p, PlayerPosition ppos, Position pos,
                TeamRoster tr, Team t, Staduim s
where pos.positionName = "Pitcher"
and s.coverType = "Dome"
and p.playerNumber = ppos.playerNumber
and ppos.positionName = pos.positionName
and p.playerNumber = tr.playerNumber
and t.teamName= tr.teamName
and s.staduimName = t.playsAtStaduimName
```

Figure 30. SQL Join Example

C.8.1. Nested Query

There are times when the result of one query is used in another. The first query may be used by the application or it may only be of interest in defining the second query. In the later case, the initial query may be nested in the second so that the result need not be returned. An example of nested query is shown in Figure 31.

```
select Player.playerName
from Player, Team, TeamRoster
where Player.playerNumber = TeamRoster.playerNumber
and TeamRoster.teamName =Team.teamName
and Team.teamName in
                (select Team.teamName
                from Stadium, Team
                where Stadium.stadiumName=Team.playsAtStadiumName
                and Stadium.city = "New York");
```

Figure 31. SQL Nested Query Example

In this example it is clear that one more join could have eliminated the nested query. In some cases the programmer may prefer to write nested queries for readability or to avoid joining many tables at one time. Joins of many tables are very hard to read and they are slow to execute. Nested joins are also a major indicator of complex data that may be better served by an object DBMS.

C.8.2. Example SQL DDL

Figure 32 and Figure 33 contain an example of creating tables in SQL that creates the Game and Promotion tables.

```
Create Table Game
(gameDateTime               date,
homeTeamName                varchar2(10),
visitingTeamName            varchar2(10),
homeTeamScore               number(8),
visitingTeamScore           number(8)
);
Alter Table Game Add (
    Primary Key (gameDateTime,
             homeTeamName,
            visitingTeamName)
    Constraint Game_Pk
    Using Index Storage (Initial 200K Next 20K)
);
Alter Table Game Add (
    Foreign Key (homeTeamName)
    References  Team (teamName)
    CONSTRAINT homeTeamFk
);
Alter Table Game Add (
    Foreign Key (visitingTeamName)
    References  Team (teamName)
    Constraint visitingTeamFk
);

Create Table Promotion
(promotionName        varchar2(10),
gift                  varchar2(10),
);
Alter Table Promotion Add (
    Primary Key (promotionName)
   Constraint promotion_Pk
    Using Index Storage (Initial 200K Next 20K)
);
```

Figure 32. SQL DDL Example (part 1)

```
Create Table GamePromotion
(
gameDateTime        date,
homeTeamName        varchar2(10),
visitingTeamName    varchar2(10),
promotionName       varchar2(10)
);
Alter Table GamePromotion Add (
    Primary Key (gameDateTime,
                 homeTeamName,
                 visitingTeamName,
                 promotionName)
   Constraint gamePromotion_Pk
    Using Index Storage (Initial 200K Next 20K)
);
Alter Table GamePromotion Add (
    Foreign Key (gameDateTime,
                 homeTeamName,
                 visitingTeamName)
   References Game (gameDateTime,
                 homeTeamName,
                 visitingTeamName)
   Constraint gameProGame_Fk
);
Alter Table GamePromotion Add (
    Foreign Key (promotionName)
   References Promotion(promotionName)
   Constraint gameProPro_Fk
);
```

Figure 33. SQL DDL Example (part 2)

C.8.3. Association Table Discussion

Many-to-many relationships must be "normalized" away in a relational model. This normalization adds association tables to represent two one-to-many associations instead of one many-to-many association. For example, the GamePromotion relation must be added between the Game and Promotion object that has a many-to-many relationship. The difference in the level of complexity of the definition code can be demonstrated by comparing the ODL in Figure 9 with the SQL in Figure 32 and Figure 33. Queries to these tables are more complex because the three tables have to be joined in order to associate the data between them.

D. Relational DBMS with Object Extensions

D.1. Informix Datablades

Informix has added to its relational database Illustra's DataBlades. Each module comes with an object model and operations to support database operations on that object. For example, the face recognition module includes operations to recognize similarity of features on images of a face. The developer has access to a specialized user interface provided with the DataBlade. The developer may submit an SQL query that may take advantage of the operations provided with the DataBlade. The following list of DataBlade modules is provided to give an understanding of the type of objects that are being supported by this technology.

The DataBlade Modules currently available for Data Warehousing include:

- DataCleanser from Electronic Digital Documents, Inc.
- FuzzIT from BBL Software GmbH
- OptiLink from Consistency Point Technologies, Inc.

The DataBlade Modules currently available for Digital Media include:

- Audio Information Retrieval (AIR) from Muscle Fish
- Face Recognition from Excalibur Technologies Corporation
- Image from Excalibur Technologies Corporation
- SceneChange from Excalibur Technologies Corporation
- Video from VXtreme, Inc.
- Video Foundation from Informix
- Visual Information Retrieval (VIR) Viewer from Virage Technologies, Inc.

The DataBlade Modules currently available for Financial applications include:

- S-Plus from MathSoft, Inc.
- Time-Series from Informix

The DataBlade Modules currently available for Geospatial applications include:

- Geocoding from MapInfo Corporation

- Geodetic from Informix
- Global/Interval from TelContar
- Spatial Database Engine from ESRI
- SpatialWare from MapInfo Corporation

The DataBlade Modules currently available for Text and Document Management include:

- Document Objects from ArborText
- Real-Time Profiling from Excalibur Technologies Corporation
- Text from Excalibur Technologies Corporation

The DataBlade Modules currently available for World Wide Web and Electronic Commerce include:

- DesCrypt from Prime Factors
- Event from Informix
- Web from Informix

D.2. Oracle 8 Data Cartridges

Oracle 8 Data Cartridges are similar to DataBlades. They can extend the database server with text, image, spatial geometry, and time series, and utilities for Web-based applications. The cartridges that are currently available are as follows:

- Spatial Cartridge includes objects that support geographic components of business information systems.
- Video Cartridge enables video and audio over different network infrastructures, including broadband networks, such as ATM, cable and satellite, and intranet networks using Ethernet.
- ConText Cartridge provides users with full text retrieval and advanced linguistic services.
- Visual Image Retrieval (VIR) Cartridge and Image Cartridge provide objects and operations, such as compression and content-based retrieval for visual information.
- Time Series Cartridge allows temporal data to be stored in the universal data server, containing functions for calendar, time-series, and time-scaling to retrieve and process data.

D.3. DB2 Extenders

IBM has had extenders for audio, image, text, and video for its DB2 DBMS since 1995. There are also modules for spatial and time-series data being developed.

E. TRAC2ES OBJECT DBMS SELECTION EXAMPLE

This appendix is included as an example of one DoD activity that evaluated their requirements, decided on an object DBMS architecture, and evaluated products within that architecture to satisfy their requirements. The list of application requirements will be very specific to each application, but this is a good example of some issues and the level of detail that may be required for selection of a persistent storage mechanism for other object-oriented applications.

The developers of the prototype for the USTRANSCOM Regulating and Command and Control Evacuation System (TRAC2ES) made the decision to use an object DBMS primarily because their data structure and the query patterns against the data were very complex. They had a requirement for fast delivery of a new system and needed a database environment that would use the C++ and Rumbaugh's Object Modeling Technique development tools seamlessly. They could not devote many assets to development of the database.

The USTRANSCOM Regulating and Command and Control Evacuation System (TRAC2ES) is used to plan and monitor world-wide movement and care of military medical patients. Data about the patient, his condition, and care are loaded into the database by the local medical treatment facility. Information about the available beds and care providers in hospitals world-wide is kept up-to-date by the local hospitals. Information about all military aircraft missions and medical facilities on each aircraft is available to the Global Patient Movement Requirements Center and the Theater Patient Movement Requirements Centers (TPMRC). TRAC2ES analyzes this complex and highly distributed data and plans missions to provide safe, speedy, and cost-effective transportation of patients to the appropriate hospital for care. The system will recommend changes if problems, such as airport closing, missed flights, and changes to hospital bed availability, occur before the patient arrives at his destination.

The TRAC2ES development team analyzed their requirements and started looking for a solution to those requirements. They were surprised to find very little support to help them make a decision. They used the DBMS Needs Assessment for Objects [2] to help them examine the key features of the commercial products against their requirements. The list of requirements for other systems may be very different but it is useful to examine their fourteen stated requirements as samples of the types of things that must be considered. The list of requirements in this appendix is taken directly from the paper by Braidic[9]. They found that no object DBMS product satisfied all of their requirements at the time of their evaluation. They chose Versant primarily based on support for the first three requirements, which they considered to be critical.

E.1. Proven Wide Area Network (WAN) Support

Effective operation in a WAN environment is critical since TRAC2ES data is distributed worldwide. They wanted a product that could technically support WAN distribution of data; they wanted a product with a proven record in supporting distributed data on a WAN.

E.2. Rich Locking Features

The planning algorithm, which creates the movement plans for patients, will touch nearly every object in the system related to airplanes, hospitals, beds, and patients during the building of its solution. Updates to this data continue to pour into the system while the algorithm is running. The far-reaching nature of the algorithm could cause it to freeze most of the database if a sophisticated locking scheme were not available.

Their application needs a wide range of locking options and explicit control over the types of locks that are applied and the conditions/times under which they are released. The locking also must be at the object level. When a hospital updates patient information, only the patient object that is being changed should be locked.

E.3. Cache Management

A critical factor in object DBMS performance is its technique for caching and accessing objects in memory after they are fetched from disk. Since the planning algorithm is accessing a high volume of data, they wanted the algorithm to ensure that the data frequently referenced would be kept in memory. The algorithm performance could not afford objects randomly swapping in and out of cache.

E.4. Location Independence

If one node of the distributed data network is down, database function at other nodes should continue operation.

E.5. Disk Mirroring

Data should be mirrored between different nodes to provide high availability.

E.6. Long Transactions

The object DBMS should allow transactions that last for hours or days, even if the user logs off the system.

E.7. Shared Transactions

Two systems may join in the same transaction to collaborate on processing.

E.8. Object Versions

The ability to create versions of an object as changes are made while maintaining the ability to reference prior versions is important in this system because of the high number of data updates and the need to maintain history.

E.9. On-line Distributed Database-wide Consistent Backup

A backup of the entire distributed database can be made from a single node without forcing any component of the system to be brought down.

E.10. Query Return When All Nodes Are Not Available

Some data is better than no data. For example, a query to return all patients in the system should not fail if one TPMRC system is down. The query should return as many patients as it could reach and inform the application that the list is incomplete as a result of a problem in the network.

E.11. Security

The data stored in TRAC2ES and the wide variety of user types that will be accessing data, require security features at the object level.

E.12. ODBC Compliance

Third-party query tools that support object DBMSs could be useful in providing additional reporting capability.

E.13. Replication

Some data in the system will need to be replicated to hosts that may have long periods during which they cannot obtain communications to the rest of theTRAC2ES network.

E.14. Private Database Support

Private databases allow the applications to manipulate select objects in an area not visible to other processes. The application can currently access objects in the TRAC2ES distributed data network and its private database.

F. OBJECT DBMS VENDORS

Fujitsu Software Corporation: http://www.fsc.fujitsu.com

GemStone Systems, Inc.: http://www.gemstone.com

Hewlett-Packard Company: odapter@cup.hp.com

IBEX Object Systems: http://www.iprolink.ch/ibexcom

Informix (Illustra): http://www.informix.com

O2 Technology, Inc.: http://www.o2tech.com

Object Design, Inc.: http://www.odi.com

Objectivity, Inc.: http://www.objectivity.com

ONTOS, Inc.: http://www.ontos.com

Persistent Data Systems: info@persist.com

Persistence Software, Inc.: http://www.persistence.com

POET Software, Inc.: http://www.poet.com

Unisys Corporation: http://www.osmos.com

UniSQL, Inc.: http://www.unisql.com

Versant Object Technology: http://www.versant.com

G. LIST OF REFERENCES

[1] Dick, Kevin, "Trajectory Analysis: ODBMS Vendors," Barry & Associates web page at http://www.odbmsfacts.com, 1997.

[2] Barry, Douglas K., R.G.G. Cattell, "DBMS Needs Assessment for Objects," version 3.1, Barry & Associates, Inc. 1996.

[3] Barry, Douglas K., "The Object Database Handbook: How to Select, Implement and Use Object-Oriented Databases," Wiley Computer Publishing, 1996.

[4] Cattell, R.G.G., et. al., "The Object Database Standard: ODMG 2.0," Morgan Kaufmann Publishers, Inc. 1997.

[5] Davis, Judith R.,. "Object-Relational Database Managers: Balancing the Investment in Relational with New business Requirements," Distributed Computed Monitor, Patricia Seybold Group, 1995.

[6] Object Management Group, "Persistent Object Service Specification," OMG Web Site at http://www.omg.org/corba/sectrans.htm.

[7] Object Management Group Web Site at http://www.omg.org/library/schedule/Persistent_State_Service_2.0_RFP.htm#RFP_Issued.

[8] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen, "Object-Oriented Modeling and Design," Prentice Hall, 1991.

[9] Braidic, Gretchen, "Mission-Critical Patient Care," Object Magazine, Vol. 5, Number 9, p.56, February 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE March 1998	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Selecting a Persistent Data Support Environment for Object-Oriented Applications	5. FUNDING NUMBERS DASW01-94-C-0054 Task Order T-S5-1446
6. AUTHOR(S) Glen R. White, Clyde G. Roby	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard St. Alexandria, VA 22311-1772	8. PERFORMING ORGANIZATION REPORT NUMBER IDA Document D-2120
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Information Systems Agency Center for Computer Systems Engineering 5600 Columbia Pike Falls Church, VA 22041-2717	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, unlimited distribution: 10 April 1998.	12b. DISTRIBUTION CODE 2A

13. ABSTRACT (Maximum 200 words)

Developers of object-oriented applications need the ability to save the state of objects. A number of relational and object database management systems (DBMSs) may be used. The commercial DBMSs can be categorized into one of four architectures: object-oriented, object-relational, object-relational mapping, and relational with object extensions. Each of these architectures has advantages and disadvantages that make it more suitable for applications with different requirements. This document discusses the major issues pertinent to selecting a persistent storage mechanism for objects. It examines the object models and database interface languages advocated by the Object Management Group (OMG), the Object Database Management Group (ODMG), and the American National Standards Institute (ANSI) X3H2.

14. SUBJECT TERMS Object-oriented Applications; Object Database Management Systems; Object-relational; Object-relational Mapping; Database Interface Standards.	15. NUMBER OF PAGES 104
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102