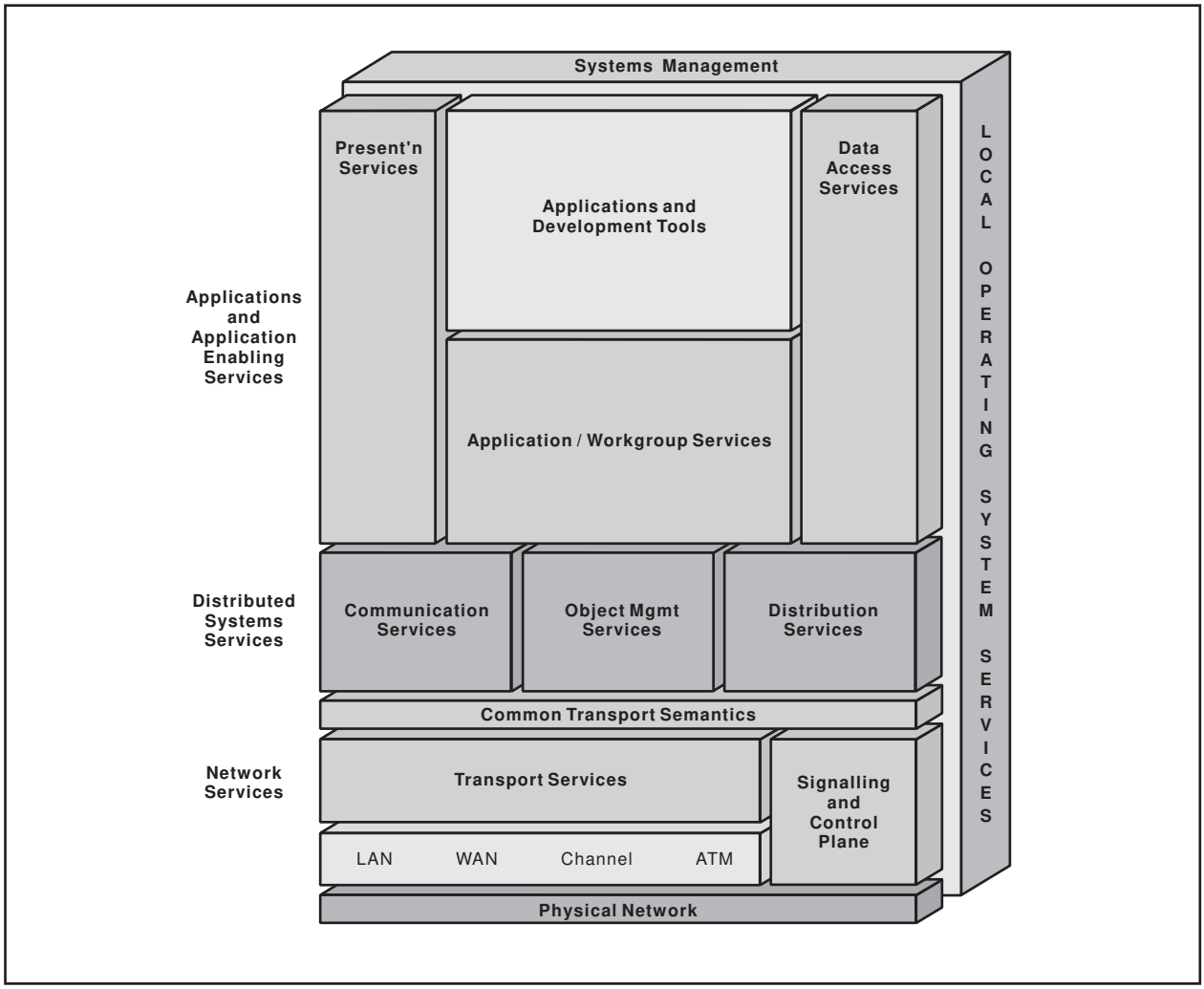
Open Blueprint

Persistence Services Resource Manager



GC23-3923-01

Open Blueprint

Persistence Services Resource Manager

GC23-3923-01

About This Paper

Open, distributed computing of all forms, including client/server and network computing, is the model that is driving the rapid evolution of information technology today. The Open Blueprint structure is IBM's industry-leading architectural framework for distributed computing in a multivendor, heterogeneous environment. This paper describes the Persistence Services resource manager component of the Open Blueprint and its relationships with other Open Blueprint components.

The Open Blueprint structure continues to accommodate advances in technology and incorporate emerging standards and protocols as information technology needs and capabilities evolve. For example, the structure now incorporates digital library, object-oriented and mobile technologies, and support for internet-enabled applications. Thus, this document is a snapshot at a particular point in time. The Open Blueprint structure will continue to evolve as new technologies emerge.

This paper is one in a series of papers available in the *Open Blueprint Technical Reference Library* collection, SBOF-8702 (hardcopy) or SK2T-2478 (CD-ROM). The intent of this technical library is to provide detailed information about each Open Blueprint component. The authors of these papers are the developers and designers directly responsible for the components, so you might observe differences in style, scope, and format between this paper and others.

Readers who are less familiar with a particular component can refer to the referenced materials to gain basic background knowledge not included in the papers. For a general technical overview of the Open Blueprint, see the *Open Blueprint Technical Overview*, GC23-3808.

Who Should Read This Paper

This paper is intended for audiences requiring technical detail about the Persistence Services Resource Manager in the Open Blueprint. These include:

- Customers who are planning technology or architecture investments
- Software vendors who are developing products to interoperate with other products that support the Open Blueprint
- Consultants and service providers who offer integration services to customers

© **Copyright International Business Machines Corporation 1995, 1996. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Summary of Changes 1

Open Blueprint Persistence Services Resource Manager 3
Introduction 3
Concepts 5
Persistence Services 6

Appendix A. Relational Database Support For Advanced Data Types 13
User-Defined Functions 13
User-Defined Data Types 13
Blob Storage and Retrieval 13

Appendix B. Notices 15
Trademarks 15

Appendix C. Communicating Your Comments to IBM 17

Figures

1. Basic Relationships between an Application, an Object, and Persistence Service 4
2. Persistence Using a Two-Level Storage Model 7
3. Persistence Using a Single-Level Store 9

© Copyright IBM Corp. 1995, 1996

Summary of Changes

This paper is a revision of the June, 1995 version. It has been revised to:

- Describe updated technology such as object caching and object distribution
- Provide additional details on handling of object relationships
- Allow objects to reflect existing application semantics
- Communicate that these services can be used by any OO programming language
- Improve the accuracy of the technical description of the service

The distinction between the persistence services deployed in a SOM environment and a native language environment has been removed.

© Copyright IBM Corp. 1995, 1996

Open Blueprint Persistence Services Resource Manager

This paper describes the technical direction for object persistence as provided by the Open Blueprint. Object persistence is a service for object-oriented (OO) applications. It provides the mechanisms for moving and mapping between the virtual storage representation and the disk-resident representation of an object.

Introduction

In the relationship between OO requesting applications and the objects they reference, persistence of the objects might not be a consideration. Some objects may be purely transient; they are created, manipulated, and destroyed within an execution scope of an application. Other objects may persist beyond the execution of an application. Still others may persist and be shared among different applications.

In many cases, it is possible and even desirable that the considerations for persistence of an object not be a concern of the requesting application. In many other cases, the application may need some explicit control over the persistence of certain parts or all of the object.

When application objects are to be permanently stored in some underlying data store, they are referred to as *persistent objects* and the services used to store and retrieve them are referred to as the *persistence service*. In this paper, *data store* means any general, persistent storage capability such as a file system or database.

The persistence service offers choices to the application and object designers. A designer can make the persistence transparent, that is, the requesting application can be completely ignorant of the persistence mechanism, if the object designer chooses to hide it. Alternatively, the object designer can expose specific persistence functions that a requesting application may need.

The data store can be any of a wide variety of implementations including relational databases, hierarchical databases, OO databases, record files, and stream files. The differences between these data stores can be hidden behind a common persistence interface. Thus, the persistence service allows OO programs to access a large collection of existing data.

The persistence service uses object designer-specified mapping definitions when retrieving or storing objects that reside in or work with traditional data stores.

The persistence service allows an OO application to be independent of the underlying data store. The application can be adapted to new or alternative data stores through the addition of alternative mapping definitions.

The choices are implemented and controlled by the application and object designers. The designers determine the characteristics of the object and its possible uses. The object designer chooses which methods will be exposed to the requesting applications and which system services will be used to assist in the efficient implementation of the persistent object. Methods controlling persistence can be some of the many methods an object can expose to a requesting application.

The persistence service is used in conjunction with other services, such as the Relational Database resource manager, Transaction Management, and Security. The persistence service, for example, does not control transaction management, but it cooperates with transaction management either directly or through an underlying data store.

© Copyright IBM Corp. 1995, 1996

The persistence service is based on the Object Management Group (OMG) standard[1] and published by X/Open. It can be implemented with assistance from IBM's System Object Model (SOM) or specific language facilities such as C++, Smalltalk, and Java.

Overall Function

The principal requirement of the persistence service is to provide common techniques for an object to be persistent. There must be a way for the system or the object designer to decide what data needs to be made persistent, and the way to store and retrieve that data.

Figure 1 below shows the relationship between the requesting application, the object, and the persistence service. The requesting application makes a reference to an object through a method call. Methods within the object can use persistence services to save and retrieve object data. There are several choices available in the way that persistence can be accomplished.

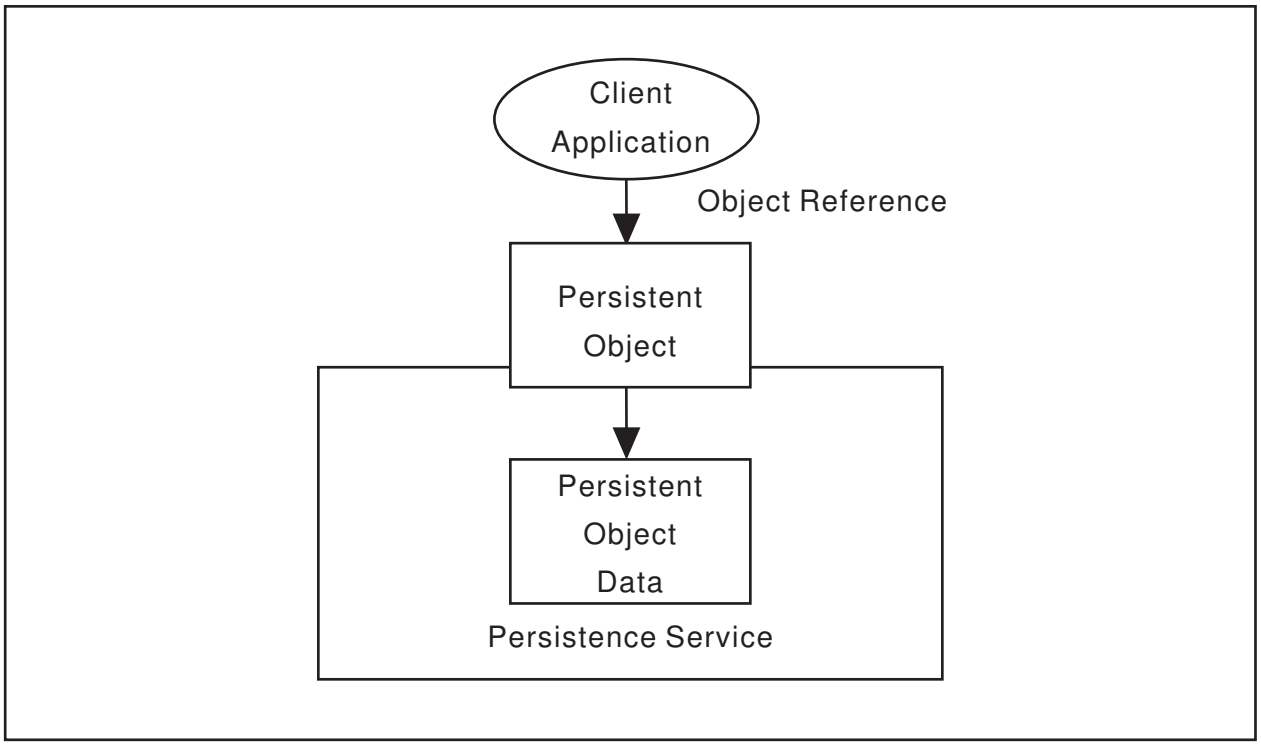


Figure 1. Basic Relationships between an Application, an Object, and Persistence Service

The instantiation of a persistent object is controlled by methods associated with the object. There is no requirement that any object use any particular persistence mechanism. For example, an object may write its data directly to a file, it may use persistence service mapping routines, or an OO development environment or OO database may be in control. The persistence service is an option for the object designer that provides capabilities useful to a wide variety of objects. By using the persistence service, the object designer need not, in many cases, write the code that maps between the virtual storage representation and the disk-resident representation of the object. The cases where the persistence service can provide the mapping code are described in later sections of this paper.

OO applications need a variety of data store types for the same reasons as procedural applications. These data stores allow a wide variety of trade-offs among resource requirements, data integrity, high availability, and different performance characteristics that fit different kinds of access patterns (for example, navigation versus query). There are several existing data store types, each with their own characteristics, capabilities, and data access languages (for example, DL/I and Structured Query

Language (SQL)), and the existing languages are the only ones available to communicate with the data stores. Either the data access languages must be adapted to accommodate OO technology, or OO "wrappers" must be used to adapt objects to the traditional data stores. Of course, the changes must allow the continued use of the data stores by existing procedural applications.

The data stores can range from an OO database to several traditional data store types. OO databases offer a closer match between the structure of an object and the stored representation of the object. Traditional data stores offer access to existing or new data which can also be used by procedural applications.

The choice of data store depends on how the data is to be used. If the data structures are highly complex and interwoven, the access patterns are highly navigational, and the data is used only with OO applications, then object designers might choose an OO database as the data store. In other situations, where the application accesses relatively simple data structures, the access pattern is non- navigational (simple or complex query), or users need to access the underlying data using both procedural and OO applications, a traditional data store would be more suitable. The data would be stored in the traditional manner for procedural access and mapped into objects for OO access.

The persistence service supports multiple concurrent use of objects in the same way a database manager supports multiple concurrent use of company data. Objects can be shared, but the object's data is protected from multiple concurrent updates.

Concepts

Transient and Persistent Objects

Transient objects are created, manipulated, and destroyed during an execution of an application program. Examples of transient objects are elements of a graphical user interface such as window scroll bars, push buttons, and pull-down menus, or temporary containers holding intermediate calculation results.

Persistent objects survive beyond the execution of an application. Object data is saved in persistent storage so that the object can be instantiated, that is, reconstructed for each execution of a using application. Examples of persistent objects are customers, orders, insurance claims, and programming design information.

Mapping between Virtual Storage and Persistent Storage

Objects are typically distinguished from one another by unique identifiers. The identifiers are used to refer to an object either by the application or by one object referring to another.

When an object is instantiated in virtual storage, a reference to the object essentially becomes a pointer, which the persistence service maps to a virtual storage address. Access through the pointer is extremely fast. This mechanism is especially valuable for compound objects, where one object references other objects. The pointers make navigation between objects fast and easy.

When objects are made persistent, the pointers become meaningless. Chances are very good that the next time the objects are instantiated in virtual storage, different pointers would be established. The relationship between objects established by these pointers, however, must be maintained. For persistence, these pointers must be transformed to a representation (an identifier) that allows reconstruction of the relationship the next time the objects are instantiated in virtual storage.

The data types associated with an object might need to be changed when the data is made persistent. Data stores typically support a limited number of data types (for example, binary, decimal, float, character

string). The data types in the application programming language might not match those supported by the data store. A mapping of data types is required between the in-memory object and its stored form. The persistence service provides tools for defining the stored schema and the mapping to a language-specific object. When the mapping is adequately defined, store and retrieve methods can be generated for accessing the object's data.

Business Objects and Data Objects

Objects can be arbitrarily simple or complex. They can represent real world entities such as mechanical assemblies, assets, or business processes, or they can simply represent a few data elements in a data store. It is sometimes convenient to classify objects into the categories *business objects* and *data objects*. A simple data object may consist of the data elements in a single data store record. The methods available in this object can consist of **SET** and **GET** for the element values in the virtual storage data object and **RETRIEVE**, **INSERT**, **UPDATE**, and **DELETE** for moving the data values (attributes) between the data object and the data store. A more complex data object can consist of the data elements in different records joined together plus the results from execution of certain functions or stored database procedures.

Business objects are usually more complex and are built from one or more data objects. They may provide additional attributes that are transient (such as an estimated year-to-date tax liability), and may provide access to new attributes that are derived from the attributes of referenced data objects. They also include the methods that implement the business logic of the applications.

The Open Blueprint Overall Approach

The persistence service provides facilities to make objects persistent. The persistence can be implicitly controlled by the system or can be explicitly controlled by the application. A common interface is provided to the application regardless of the underlying data store. The persistence service is applicable to both business objects and data objects. The persistence service works with other Open Blueprint resource managers to allow the necessary authorization, security, transaction, and distribution controls over the underlying data.

The persistence facilities are described in more detail in the following sections.

Persistence Services

The persistence service provides a choice of programming models for using persistent objects. The choice is between a single-level store or a two-level store programming model. The major difference between single-level store and two-level store is the visibility of the persistence controls to the requesting application.

Two-Level Store

In a two-level store programming model the requesting application is aware of an object's persistence requirements and it explicitly controls the movement of data associated with individual objects between virtual storage and the data store.

In a simple usage of the two-level store, the requesting application identifies the object to be retrieved with an object identifier and issues a retrieve request to the persistent object manager. The object is then instantiated in virtual storage.

Applications written for this model deal with objects, but also have access to the facilities of the underlying data store. This provides more flexibility in accessing the data. A mix of native data and object services is allowed, for example, the application may use object query services but also rely on the data store concurrency and authorization controls. Object designers and even end users may see the underlying data models (tables, records) as well as objects. They can then use underlying data manager functions such as an SQL query against a relational database. The SQL call-level interface (CLI) can be used through an OO wrapper to the CLI function calls.

An example of the need for this flexibility is when an existing application is partially converted to gain some benefit of OO technology. It may be non-transactional and written against a file system. The application logic is already in place for deciding what portions of the object data need to be stored and when in the processing it is appropriate to do so. Storing all of the object data at one time may not be necessary. For migration and performance reasons, the application needs explicit control over the data store.

Figure 2 below represents an overview of how the persistence two-level store service interfaces to the application and the various data stores (relational database, hierarchical database, and record files).

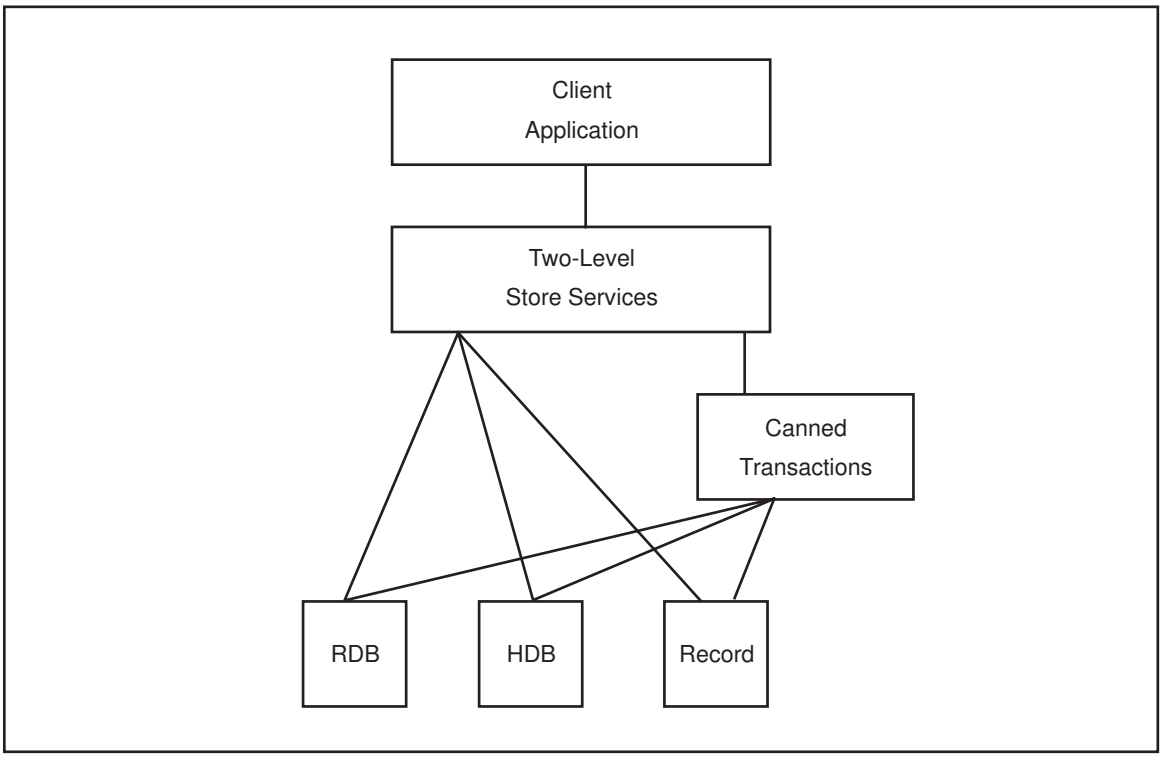


Figure 2. Persistence Using a Two-Level Storage Model

All objects are designated by either an identifier or a cursor. Specific objects are designated by an identifier that determines where the data associated with that object is stored. Collections of objects are designated by a cursor that is used to move from one object to another in the collection.

A mapping class provides a way to get data into and out of an object. A mapping class has the responsibility of mapping between the object world and the storage world. It plays critical roles in identifying the data store as well as providing convenient and efficient access to it. The mapping class is language specific. Objects are rendered to the application in a form consistent with the application programming language, either as a native language object or a cross-language SOM object.

Methods (**RETRIEVE, INSERT, UPDATE, DELETE**), are provided for control over the materialization and storage of data objects. The requesting application deals directly with the persistent data store, albeit in an OO style. The representation is usually the same as that of the underlying data store, that is, the mapping between persistent object data attributes and the data structure in the underlying data store is usually 1-to-1. However, compound objects can also be supported, where objects of one type may contain references to objects of other types. Business objects, in addition to their unique business-oriented methods, can inherit the data manipulation (persistence) methods from the associated data objects.

As illustrated in Figure 2 on page 7, object-to-data store mapping can also include the use of additional processing by stored procedures or "canned transactions," for example from CICS or IMS. In this case, it is the result of the additional processing that is mapped into the object. Support for mapping objects via canned transactions is valuable for capturing the application semantics associated with the underlying data.

The data stores are database managers or other facilities that provide data storage through specific access methods. The data stores take responsibility to ensure persistence and recoverability.

The runtime class library also supports query results. In a query where multiple rows are retrieved, they are returned as a collection of data objects, and a cursor class is provided to iterate through the collection. For more complex mapping configurations, or more flexibility, the mapping definitions can be used with a generalized run-time class library for dynamic data access. This approach is the most flexible, and can handle all mapping configurations. Arbitrarily complex SQL statements can be built at run-time to perform complex queries and other data manipulation operations. A dynamic data object (DDO) is used. The DDO is a generalized mechanism for accessing an object's data and the data type. It is constructed at runtime. Because of this, the DDO provides support for dynamic data typing, which allows a requesting application to deal with data whose data types are not known at compile time. This becomes useful when designing objects based on a query result where the application cannot determine the scope and style of a data access operation in advance.

Mapping is controlled by predefined schema mapping information provided by an object designer. See “Schema Mapping” on page 10 for more details on mapping.

For relational data stores, methods are also available to access the relational catalogs.

Single-Level Store

In a single-level store programming model the requesting application deals with all objects as though they are directly available in virtual storage, and can navigate among objects using language-specific object references. Other than transaction management and object creation, object persistence is transparent to the application; no explicit operation is needed to store or retrieve the object. An object reference from the requesting application is all that is needed to instantiate the object in virtual storage. The persistence service automatically issues appropriate updates to the underlying data store at a transaction boundary (implicit or explicit commit) if any part of the object has been changed.

Object query (see “Object Query” on page 11) can be used to select which object or collection of objects is to be used initially by the application. These objects can then provide references to other related objects.

All code needed to move data between the application's virtual storage and the persistent store (and to transform this data when necessary) is performed by the persistence service and the data store. In order to achieve transparency, the persistence manager must intercept object references and provide efficient, high-performance transfers of objects between virtual storage and the persistent store.

Within the single-level store approach, the persistence service provides two main implementations for movement of object data between the application's virtual storage and the persistent store. The implementations are:

- Object storage in a traditional data store (using two-level store methods to control the data store)
- Object storage in an OO database

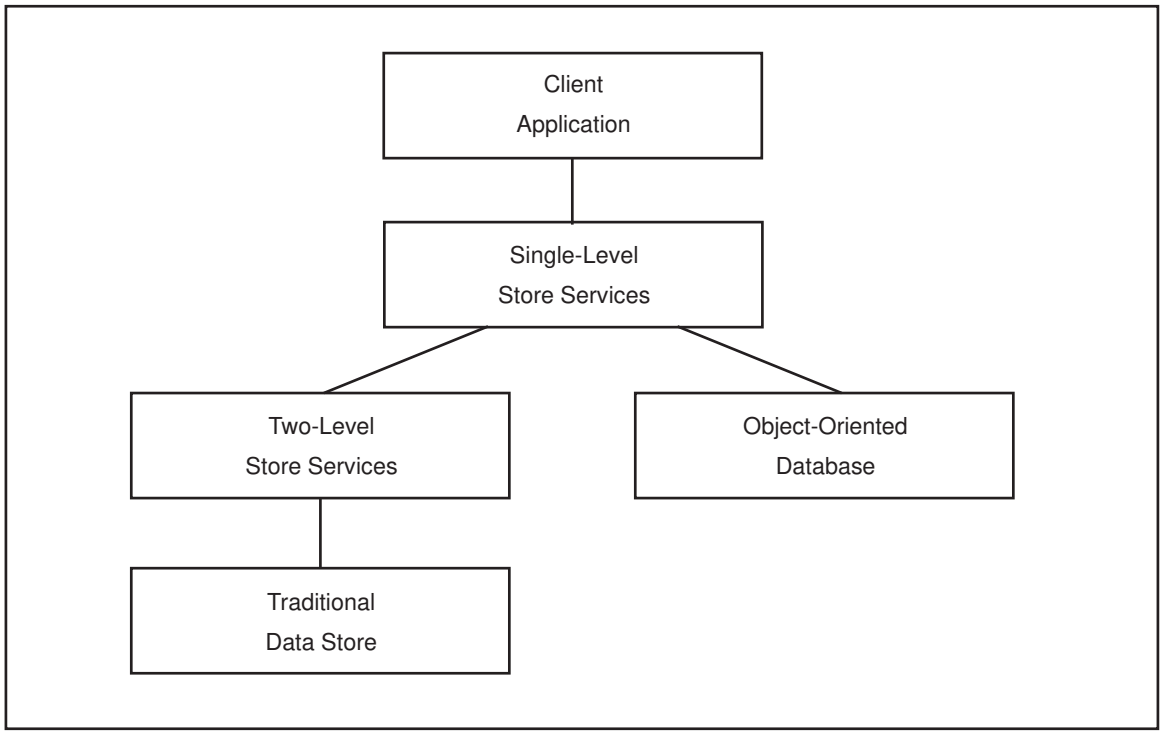


Figure 3. Persistence Using a Single-Level Store

For single-level store environments, performance is enhanced by maintaining instantiated objects in a cache on the application host. Object integrity is typically maintained by keeping the cache synchronized with the data in the underlying data store. When the application ends a transaction with a commit operation, any updated persistent objects are written back to the data store, locks are released, and the cache is invalidated. When the application ends a transaction with a rollback operation, updates are discarded, locks are released, and the cache is invalidated. For some application environments, objects may need to be retained in the cache across transaction boundaries. In this case, some application-specific logic is needed to deal with consistency between the objects and the underlying data store. This situation might occur when an application is working with a large number of objects and is updating only a few of them at a time. For example, in a process-control application, the program may need to calculate and commit changes in a few process-control objects, but would not need to subsequently retrieve fresh copies of the entire set of process-control objects in the cache.

Caching Considerations

A separate cache may be maintained for each application process. Concurrency control of the underlying data is maintained in the data store by appropriate locking protocols. Alternatively, the cache may be shared across multiple application processes. In this case, concurrency control is provided at the object level, and at the data store level.

As a performance improvement, the persistence service can provide for pre-fetching of objects into the cache. Pre-fetching allows data objects to be brought into the cache prior to their use by the application. When these objects are subsequently referenced, they can be accessed without additional I/O operations.

This is an object designer's choice and the parameters specified for pre-fetching can affect the underlying operations such as the type of query sent to the data store manager.

When an OO database is used, the OO database stores this data in its own private persistent store, transforms pointers when necessary, and arranges to be invoked when necessary for data movement (that is, on commit or to retrieve a referenced object). When an object reference is made, the OO database manager first checks the cache to see if the object has been instantiated there. If not, the database manager retrieves the object (or perhaps multiple related objects as discussed above) from the persistent store and instantiates the object(s) in the cache. From the cache, the object is then made available to the application.

Single-level store capability for objects based on traditional data stores is accomplished by providing a facility that shields the class designer and client application from any of the underlying two-level store activity. The application obtains single-level store benefits by using this facility for object references. When an object reference is made, the facility first checks the cache to see if the object has been instantiated there. If not, the facility uses the two-level store services to obtain the object data (and perhaps data for related objects) and instantiates the object in the cache. From the cache, the object is then made available to the application.

The facility stores any new or changed object data when the application requests a commit operation.

Object Relationships

In some system configurations, relationships between objects can be mapped and enforced through facilities in the data store. For example, if two data objects represent different records in a database, and a relationship between the objects is represented by a reference from one object to the other, that relationship can be represented in the database by a foreign key in one record matching the primary key of the other. This mechanism preserves the relationship between the objects and enhances performance when the objects are retrieved.

As another example, employee and department object classes may have a relationship (all employees belong to one department) and each object class can be mapped to a relational table in a database. The referential integrity rules available in the database can thereby be used to enforce relationship integrity between the objects above the data stores. In this case, an employee object cannot be stored for a department that does not exist in the database.

Schema Mapping

When object data is to be stored in a traditional data store, mapping is required between the data of a persistent object and the data in the underlying data store. Mapping information can be gathered from the object designer, from application program object specifications, and from the data store schemas. (A schema describes the data structure of a file or database. It defines attributes such as field length and data type for each data element.) The mapping information is stored in a mapping language created and interpreted by mapping utilities.

A set of graphical tools are provided to help the object designer specify the mapping between objects in virtual storage and the data store. A set of schema mapping configurations are supported. The mapping information deals with how objects relate to the underlying data. It also deals with how objects relate to each other as discussed above, for example, how employee objects relate to department objects. Foreign keys can be represented as reference attributes, to allow object-based navigation and query.

The schema mapper provides flexible mapping capabilities. The object designer can specify whether an object is mapped to one or more rows in a relational store, segments in a hierarchical database, records in a record-oriented file, or input/output parameters for canned transactions. Mapping can be simple and

automatic such as mapping a complete schema to an object, or mapping can be controlled so that only specific elements of a database are mapped to an object.

For the less complex mapping configurations, the tools can generate data access methods for the object class. The mapping operation also includes the routines necessary to interpret the mapping language, obtain the data from the data store, perform any necessary data conversion, and instantiate the persistent objects. These methods are the most efficient at runtime, because the mapping requirements have been predetermined and bound to the application. In particular, for IBM relational databases, the tools can generate static SQL which has already been parsed and optimized for efficient data store access.

For more complex mapping configurations, or for more flexibility, the mapping definitions can be used with a generalized runtime class library for dynamic data access. This approach is the most flexible, and can handle a broader range of mapping configurations. Mapping methods can also be written (hand-crafted) to obtain arbitrarily complex mapping operations.

The mapping methods are language specific. They render objects that are consistent with the host application programming language environment (for example, SOM, native C++, Java, or Smalltalk).

Object Query

An application can need to inquire about the attribute values of, or even the existence of objects that meet certain conditions. The objective of an object query is to analyze identified objects based upon some query criteria and return (references to) the subset of objects that meet the criteria.

In an object query, one or more object data or reference attributes are analyzed. The objects can be contained in one or more input collections. The result is a collection of references to objects of one type, and a cursor to iterate through the collection. The objects in the returned collection can be the set of objects in one of the query input collections, a subset of those objects, or a set of new dynamic and transient objects formed from a projection, join, union, or function executed against the input object collections. In any case, the objects can contain references to other related objects.

In general, a query is sent to a query interpreter, passing the query expression as a parameter. If the underlying data stores have their own query facilities, the object query interpreter can decompose and modify the object query into segments that the data stores can process. When the query results are returned, the object query interpreter is responsible for composing the result and handling any portions of the query that could not be delegated. If the underlying data stores do not have query support, the object query interpreter must do all the necessary work to obtain query results.

The object query service uses the object schema information and the schema mapping information in the underlying persistence services to determine what parts of the object query can be pushed down to the data stores.

Objects in a Network

Objects can be based on data that is either local or remote from the host where the application is running. The schema mapping operation determines which underlying data store is used for making the objects persistent. The full power of a distributed database can be used to assist in basing objects on remote data. This includes distributed access to heterogeneous (multiple vendor) database managers.

Full distributed object support is available through the Open Blueprint Object Request Broker. The ORB allows an object on one host to invoke methods on an object on another host. The application developer need not be concerned with the location of the object or the underlying data store. Instantiating the object in virtual storage is handled by the system services beneath the Application Programming Interface (API).

Migration and Coexistence

A major advantage of the persistence services design is that it allows access to traditional data stores as well as OO databases. It allows new OO applications to coexist with new or existing procedural applications. Both kinds of applications can access the same enterprise data base. New applications can be developed without the need to migrate the enterprise data to a new OO database. The migration path to OO technology can be taken in shorter planned steps instead of having to migrate applications and data all at the same time.

Relationship to Other Open Blueprint Services

The persistence service is used with other Open Blueprint services. The Relational Database, Hierarchical Database, Object-Oriented Database, and File resource managers can be used as the underlying data stores for persistent objects. The persistence service cooperates with the Transaction Management resource manager either directly or through an underlying data store. The Access Control resource manager can be used to control access to persistent objects.

---

[1] OMG Document Numbers 94-1 and 94-10-7: *Common Object Services Specification*, Chapter 6, "Persistent Object Service Specification". The specification allows implementation flexibility in the use of specific components such as the Persistent Data Service, any of several datastore protocols, and the Persistent Object Manager.

Appendix A. Relational Database Support For Advanced Data Types

In mapping to certain relational databases, one can exploit features that allow easier mapping and reduce the language impedance between an OO programming language and a traditional data store. Many relational databases now have extended functions and data types that allow a better match (more direct mapping) between an object and its stored representation. The most important features are briefly described here.

User-Defined Functions

This support allows encapsulation of data and methods to produce derived results. A programmer can write the function to access data from the database, perform calculations and aggregations on the data and return a scalar result. A simple example is calculating a person's age from a date-of-birth column and today's date.

User-Defined Data Types

This support provides strong data typing between stored data and user-written functions. The data typing is enforced by the database engine. Strong data typing reduces the risk of logical errors in an application. For example, a trigonometric function written to accept arguments expressed in degrees could not be invoked using arguments defined as radians where degrees and radians are user-defined data types. Conversely, if there are two equivalent trigonometric functions, one for degrees and one for radians, the database manager will ensure the selection of the correct one.

Blob Storage and Retrieval

Binary large objects (blobs) are used to store objects made up of non-formatted data when the size of the object is larger than the normal maximum size of a base data type. The blob is stored as a user-defined data type derived from a character string base data type.

Examples of these types are the multimedia types of audio, image, and video. In the cases where a blob is too large to store within the database, or when it would be inefficient to do so, the blob is represented in the database by a unique object identifier and the actual blob is stored in a separate location (for example, a video server).

Additional functions are provided in the blob support to allow retrieval and manipulation of parts of a blob in a manner similar to substring and concatenation operations.

© Copyright IBM Corp. 1995, 1996

Appendix B. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

CICS
IBM
IBMLink
IMS
Open Blueprint
SOM
System Object Model

The following terms are trademarks of other companies:

C++	American Telephone and Telegraph Company, Incorporated
Java	Sun Microsystems, Inc.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the U.S. and other countries.

© Copyright IBM Corp. 1995, 1996

Appendix C. Communicating Your Comments to IBM

If you especially like or dislike anything about this paper, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply. Feel free to comment on specific error or omissions, accuracy, organization, subject matter, or completeness of this paper.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

- If you prefer to send comments by FAX, use this number:

  United States and Canada: 1-800-227-5088.

- If you prefer to send comments electronically, use one of these ID's:
  - Internet: **USIB2HPD@VNET.IBM.COM**
  - IBM Mail Exchange: **USIB2HPD at IBMMAIL**
  - IBMLink: **CIBMORCF at RALVM13**

Make sure to include the following in your note:

- Title of this paper
- Page number or topic to which your comment applies

© Copyright IBM Corp. 1995, 1996

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC23-3923-01