More Effective Design with Mappings p.28 • Criteria for QA p.6

JOOP

The Journal of Object-Oriented Programming

The Global Authority on Object Development

October 1997 Vol. 10, No. 6

Ensuring successful

Mentoring & Training

- Your Updated Guide to 280+ Providers
- Tips for Maximizing Your Investment
- Keys to Java Migration

Andrew Koenig
Closing in on the C++ Standard

Won Kim
RDB "Universal" Servers Giving Real ORDBs a Bad Name

LaLonde & Pugh
Generating Speech with Smalltalk

Evaluating Persistent Object Systems

Reviving Functional Decomposition in OOD

US $9/Can $12/Intl $13

Get Ready for **C++ World,** San Jose, November 3-7 • p.14

Kalyani Chennupati and Hossein Saiedian

Department of Computer Science, University of Nebraska at Omaha
Omaha, Nebraska, 68182-0500
email:hossein@cs.unomaha.edu

An Evaluation of Object Store Management and Naming Schemes in Persistent Object Systems

ABSTRACT

Persistent objects survive a program activation and outlive the application that creates them. Other applications may make use of persistent objects either just to use them or to manipulate them. In such a case, a mechanism is needed to allow access to the existing passive persistent objects. A naming mechanism allows access to the persistent objects so that they can be retrieved between applications or between sessions of the same application that created them. The object store organization that is used to manage the objects in the persistent store is also important because it influences the accessing of the persistent objects. This article discusses the various mechanisms for accessing and retrieving persistent objects used locally or remotely. The different Object Store Organizations used in some persistent environments are also dealt with. The different schemes used for Persistent Object Store organization and accessing and retrieving objects are discussed with respect to their basic concepts, their merits, and their weaknesses, if any. The discussion will be based on comparison and the choice of one scheme over the others in a persistent environment. There is no one unique choice based on the schemes and therefore the article will deal with different persistent environments, their underlying principles, and the scheme that best fits each environment. Pointer swizzling, another important area of study in persistent systems, is also briefly discussed concerning the issues and the importance of swizzling. We evaluate the naming and object store management schemes in different persistent environments.

The lifetimes of data or objects in a traditional programming language do not extend beyond the activation of the program that created them. On the other hand, persistent objects survive a program activation and outlive the application that creates them. The real world can be modeled in programming languages using transient memory, whereas it can be modeled in databases using persistent memory. That is, the data that are associated with the program variables do not survive past the execution of the program that created them, whereas the data that are associated with a database typically survive long past the execution of the program that created them.[18] The persistent abstraction is designed to provide an underlying technology for long-lived, concurrently accessed, and potentially large bodies of data and programs. A few examples of such systems are CAD/CAM systems, office automation, CASE tools, software engineering environments, and object-oriented database systems. In an object-oriented database environment, the objects that are created also fall under the category of persistent objects. Persistence, in general, means that certain program components survive the termination of the program. These components, therefore, have to be stored permanently in secondary storage that is also called the persistent store.

In a database environment we can distinguish two different components that can be made persistent. These are objects and variables. In some object-oriented data models, like the GOM, Sort, or Object Types can also be made persistent.[9] In this article we consider the objects as being provided with persistence. The persistent objects may be created by an object-oriented database. Once the persistent objects have been created, they are stored in the non-volatile memory or the persistent store. Other applications or other sessions of the application that created the objects may make use of these objects either to make use of their values or to manipulate them. These persistent objects must therefore be retrieved from the persistent store in order for an application to make use of them. This retrieval can be done by using a naming mechanism or by using some linkage mechanism. Therefore, the design and implementation of the mechanisms that support persistence must also address the important issue of how to retrieve a persistent object.

Another issue that is important from the perspective of retrieving objects is pointer swizzling.[9] Pointer swizzling is a technique that is used for optimizing the access to persistent objects that are resident in main memory other than in the secondary memory or persistent store. The persistent pointers in the form of unique object identifiers are transformed or swizzled into main memory pointers or addresses. Although a detailed discussion of

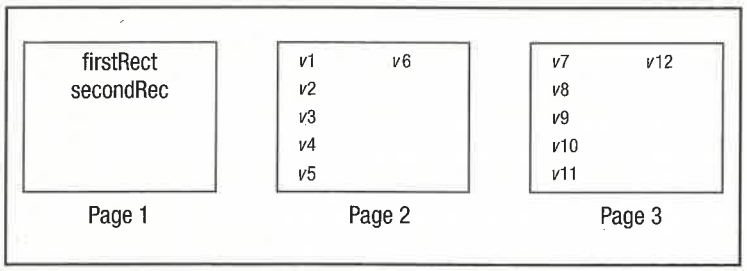

Figure 1. Physical clustering.

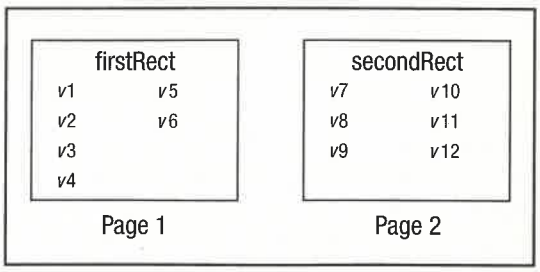

Figure 2. Logical clustering.

the different techniques of pointer swizzling is beyond the scope of this article, a brief overview of the concept is called for.

In a persistent environment, an object could refer to another object or objects. A persistent environment could also involve the use of complex objects where an object is composed of one or more objects. This highlights the importance of the issue of how these persistent objects are stored in the persistent store. The way in which the persistent objects are stored in the persistent store influences the speed with which they can be retrieved. Therefore, the storage of the persistent objects in the persistent store is also an important area of consideration in a persistent environment.

In this article we deal with the persistent storage organization of the objects, and the retrieval mechanisms of the objects from the persistent store. Pointer swizzling, which is a measure to optimize the access to persistent objects in main memory via converting references to main memory pointers, is also discussed briefly. There are a number of persistent environments/architectures that use one technique or the other. Therefore the area of persistent store organization and the retrieval of the objects from the persistent store is not a closed one with respect to the existing mechanisms available. The purpose of this article is to emphasize the importance of the persistent store organization and the retrieval of the objects. These two important areas in a persistent environment are discussed by using a few persistent environments as the basis.

The rest of the article is organized as follows. The section titled "Persistent Object Store Organization" deals with the persistent store organization and describes the different mechanisms used in a few persistent environments. The section "Accessing and Retrieving Objects" deals with the retrieval mechanisms of the objects from the persistent store. The section "Pointer Swizzling" deals with the concept of pointer swizzling. The section called "Evaluation" presents the discussion of several schemes used for object store as well as for retrieving the objects and analyzes their advantages and/or pitfalls. The final section concludes the article.

PERSISTENT OBJECT STORE ORGANIZATION

An application that needs a persistent object has to finally obtain it from the persistent object store. In an environment that uses persistent objects, an object can refer to other objects. When an application program refers to an object which in turn refers to other objects, the object store organization plays an important role in the speed of retrieval of the persistent objects. A mechanism that clusters related objects can lead to faster retrievals. IK[16] and the O2 system[2] use such a mechanism to organize their object store for faster retrieval. Objects, in the case of complex objects, can be grouped together according to their physical characteristics, i.e., objects that are instances of the same class can be grouped together. On the other hand, objects can also be grouped based on logical considerations. Complex objects and their referenced objects can all be grouped together. Consider the objects firstRec and secondRec of type rectangle referring to objects v1, v2, v3, v4, v5, v6 and v7, v8, v9, v10, v11, v12 respectively of type vertex. Grouping can be physical (Fig. 1) or logical (Fig. 2).

The objective of clustering is to group the objects that are co-referenced in physical memory. This grouping should eventually ensure the minimization of the number of I/O operations, which is critical for data intensive applications like CAD/CAM applications. The object management systems in a persistent environment have to ensure effective management of large numbers of objects. Many of the complex object database systems support the concept of object identity and object identifier. A navigation operation involves following the object identifier to access the referenced object. Navigation is therefore an essential operation when dealing with complex objects. The navigation operation is difficult to implement efficiently because every navigation operation inherently causes one disk access operation. This increases the access time of the complex object. This inefficiency is due to the fact that logical connections among the objects are completely independent of the physical organization of the objects in general, and the navigation operation among those objects causes a lot of random accesses on persistent storage devices.[8]

Consider a complex object rectangle that has six objects of type vertex, a length variable and a breadth variable. Thus, a complete description of a rectangle consists of seven objects. If the rectangle object with id1 and the vertices with id2, id3, id4, id5, id6, and id7 are scattered throughout the pages on the disk, then a total of seven disk accesses are required to fetch the complete rectangle representation into main memory. If all the objects reside on one page, then only one page access is needed. This leads to a savings of a factor of six. Therefore, the savings with respect to access time increases when the logically related objects fit into a single page. Besides, less buffer space is wasted in main memory and the percentage of objects on a page needed by an application is high. Clustering those objects that are accessed together

in an application increases this percentage and, hence, increases performance. The main motivation behind the clustering of objects is, therefore, improved performance during access of logically related objects.

Object-oriented databases that rely on physical object identifiers commonly apply a simple clustering mechanism that exploits user hints.[9] The user can indicate where a newly created object is to be placed when using this strategy. This is mostly done by allowing the user to specify another object in whose proximity the newly created object is to be stored. This is a disadvantage because the user has to solve the complicated clustering problem. Besides, if the object in whose proximity the newly created object is to be placed is stored on a page on which the newly created object does not fit anymore, there will be no control over the clustering.

A clustering mechanism should provide a means to place logically connected objects as close to each other as possible on disk memory so that the number of disk accesses is reduced. In the following subsections we discuss the various methods that are used for clustering in various persistent environments.

Clustering strategies in the O2 system

The O2 system is set up as a server and a network of workstations. The object manager that manages the persistent objects is built on top of the Wisconsin Storage System (WiSS).[4] In O2 objects are uniquely identified by object identifiers. The objects are mapped to records and the records, in turn, are identified by unique Record IDentifiers. The Record IDentifiers describe the physical address of the records. WiSS, which provides the basic support for persistence, allows records to be stored near one another, thereby offering the possibility of clustering.

In O2, the clustering mechanisms rely on inheritance and the structural information given by the object types. All the components of the object are not clustered together. Instead, grouping strategies are defined which take the operations performed in the database into account. The O2 system uses clustering strategies based on static clustering, which are based on the concept of placement trees.[2] The type structure of a class in the O2 system may be represented as an infinite tree. A placement tree is any finite sub-tree that is extracted from this infinite tree. A clustering strategy for the system classes is a set of placement trees. For a particular class hierarchy, there could be a number of placement trees based on which objects call which other objects. The best placement tree is to be chosen to have the best possible clustering of complex objects. Clustering is transparent to the user of the O2 system. Thus, the placement trees can be modified at any time without affecting the programs.

Consider a complex object of type Car that references objects of type Tire, Engine and Parts, where Tire further references an object of type Material. The objects are placed in logical clusters using the placement trees associated with the class of which they are instances as shown in Figure 3. O2 assumes that the size of the logical clusters is unlimited and is composed of a root object and a set of objects grouped with it. As the objects are mapped to records in O2, the physical clusters are composed of a set of records. The physical clusters are mapped to a memory page. For a given class the best placement tree has to be found so that the clustering is the most effective one. To do this, the most frequently performed operations in the database are taken into account. O2 uses a clustering algorithm that implements the complex objects with inheritance. The clustering algorithm is a greedy pattern matching algorithm with no page splitting.

Clustering in the IK system

IK is an object-oriented platform, the main aim of which is to simplify the development of distributed and persistent applications. In IK all objects are created and accessed in a single and uniform manner. IK follows the general architecture and model as defined in ESPRIT project Commandos.[3] In IK, the object space includes all the objects in the persistent store and all the other objects that exist within the application address space. The global names are assigned to objects when their references are sent between applications. The objects are grouped in such a way that object graphs cross address spaces under a single global name. IK assigns objects to clusters at runtime and hides, as much as possible, the object's graph under a single global name.

Objects are assigned to clusters such that clusters have only a single globally known object called the head object. The body of the cluster is composed of all the objects that are only reachable through the head object. Therefore, the whole cluster is identified by a global name of its head object and the objects belonging to the cluster body do not need a global name because they are only referenced within the cluster. In IK the persistent

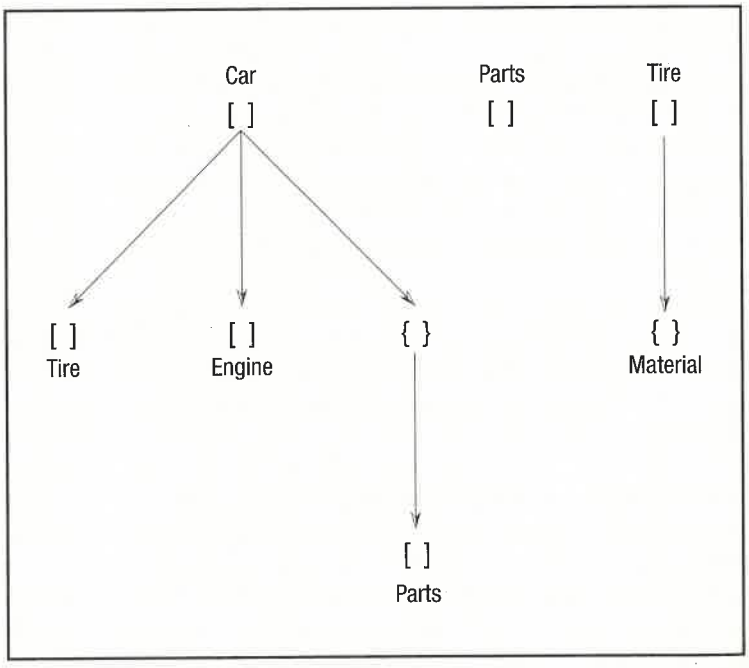

Figure 3. An example of a Placement Tree.

store is supported by a set of storage servers. A generic runtime library (GRL) linked with each application cooperates with the persistent store to support the global object space. The GRL keeps track of all global names known by the application in a table that is called the globally Known Object Table (KOT). Initially, each object in the KOT is considered to be the head of the new cluster. The graph of each object is traversed. The visited objects are tagged with the mark of the head object. If an object is tagged by different marks, it is promoted to a new cluster head. After making all objects reachable from the KOT with a cluster head mark, objects with the same mark are grouped together.

Clustering in Arjuna

Arjuna is an object-oriented programming system that permits the creation of fault-tolerant, distributed applications.[6] Arjuna uses different classes for its programming. The class ObjectState is responsible for maintaining a buffer into which the instance variables constituting the state of the objects can be saved. To manage and locate ObjectState instances, Arjuna employs an object store called Kubera. This object store is implemented by a class called ObjectStore. The ObjectStore class is, hence, responsible for the organization of the object states in persistent store. The ObjectStore class uses a combination of two approaches for the actual organization. One approach considers that all objects could be maintained in a contiguous portion of persistent store. The other method considers clustering objects to improve the speed with which a set of common objects is activated.

The identity of an object is considered a function of the class of the object. The name of the class can, hence, be used to structure the object store organization. Arjuna uses a method where objects that are instances of the same class can be maintained together. The name space of the object store can, therefore, be partitioned, which improves the speed to locate a passive state of an object. Complex objects can also be retrieved by this mechanism. For complex objects Arjuna uses the concept of implementing the entire logical object store as a collection of individual object stores, one for each class of objects, each containing all the passive states of instances of that class. If each ObjectStore is a persistent object, then all objects and classes will be reachable using the object store that contains the passive states of the object store objects.

The class ObjectStore is, hence, responsible for maintaining the physical location of all ObjectState instances in persistent store. Each ObjectState instance is a persistent object whose passive state is maintained by another ObjectStore instance that is considered to be the root of the object store. To locate an object, the object store for the class of that object would be located using the root object store. This scheme enables users to create their own object stores to manage any classes that they may define.

Other clustering strategies

Kato and Masuda[8] have proposed a scheme to notably accelerate the navigation operation among a sea of complex objects by increasing the number of effective objects per disk page. They use a scheme where some attribute values of the referenced objects are kept cached within the page in which the referencing object is placed. According to the authors, the navigation from the referencing object to the referenced objects requires only one disk access. This scheme, called persistent caching, is an implementation technique for complex objects with object identity. The authors performed an analytical performance evaluation that showed a significant performance gain in navigation using this scheme.

Shannon and Snodgrass[14] have introduced a new approach to clustering in a persistent object store called semantic clustering. The clustering is to be specified by the programmer. The mechanism that they propose uses clustering by partial closure, i.e., a subset of the attributes of the referenced objects is used for clustering. The objects are then fragmented in a controlled fashion by splitting the attributes across page segments. The ordering of segments on persistent store is used to reduce I/O times when accessing the referenced objects. The authors show that by using the tools and environment that they used, semantic clustering is faster than other forms of clustering by 20 to 35 percent. This kind of clustering mechanism has its greatest disadvantage in the fact that the programmer has to specify the clustering, which can become very tedious. Besides, because the scheme uses fragmentation for the objects, more disk space is required to store cross pointers between the segments.

ACCESSING AND RETRIEVING OBJECTS

A system that supports persistence has to provide a mechanism by which the users of the system can name, and subsequently access an existing persistent resource. The simplest way of doing this is through the use of the file system naming mechanisms used in traditional operating systems like UNIX, where the persistent resources are the files. The symbolic names, which are character based, are used to access the files in the UNIX operating system. In a similar fashion, objects that are persistent resources can be accessed by symbolic names at the user level or the application level. These symbolic names are to be mapped to the system level names to finally access and retrieve the objects in the persistent system. Arjuna uses such a naming scheme to refer to objects that are created using a tool it provides. IK uses different levels of object naming. The global names of the objects are used to find the objects on the persistent store. Persistent objects can also be accessed by links or references. The underlying concepts of these two schemes for accessing persistent objects are discussed below.

Object naming

To access a persistent object that is stored in the persistent store in a persistent environment, an application can make use of symbolic names. These symbolic names are then to be mapped to the system level or the architecture level names to finally access the objects. We discuss two naming schemes that are used in different persistent environments. The naming mechanisms in Arjuna and the IK system are detailed below. Both the systems are for distributed and persistent applications.

Object Naming in Arjuna. Arjuna adopts a three part naming

scheme, as described below, which enables the persistence mechanism to locate an object with a network of nodes. In Arjuna, the persistent resources are instances of classes that are constructed using the tools it provides. The passive forms of the objects that are constructed therefore have to be retrieved from the object store in order for the application to use them. To provide an acceptable user interface to an application that accesses persistent objects, symbolic names can be used at the user level that can be mapped to the system level naming mechanisms. Arjuna uses a scheme where a fully quantified name of an object can be considered to consist of three parts: instance name, class name, and node name. The instance name is the object in which the application is interested. The class name forms the class of which the object in question is an instance. The node name specifies the node in the distributed network where the object is actually stored. The users of the system do not have to be aware of this fully quantified naming scheme. A name server could map the partially quantified names supplied by the user to an equivalent fully quantified name.

Arjuna provides a class called ArjunaName that supports a string based naming scheme. This class is provided to the user of the system. To identify an existing persistent object, an instance of ArjunaName has to be declared which is then used as an argument to the actual object instance declaration. An example of this is shown below. In this example, an existing instance of the class WorkSheet called FileDetails is accessed through the identifier ClearSheet in the application:

```
WorkSheet ClearSheet (new ArjunaName (FileDetails));
```

According to this scheme, if the persistent object does not exist, then it is created at the node in the system where the application is running. In a situation where there are multiple objects existing on different nodes with the same name, then any one may be selected. To access a specific persistent object, the name can be further quantified. The following code is used to access an instance of the WorkSheet class on a node called bayou:

```
WorkSheet ClearSheet (new ArjunaName (FileDetails, bayou));
```

However, this scheme will require the user to know on which node in the system the object in which he is interested is stored.

The class ArjunaName in Arjuna implements a mapping mechanism that relies on access to a name server. The name server maintains all the information to locate the object. The partially quantified names supplied by users of the system are used to locate the object. Before doing so, the ArjunaName object fills the fields for the fully quantified names that were not specified by the user.

Object Naming in the IK System. In IK, which is a distributed and persistent environment, every object can be referenced by every other object in the system uniformly. Because of this, unique and long lived names are required to access and retrieve the objects. The IK platform includes a Name Server to assign and deassign the user-defined names, which are readable character strings, to global names. The collection <user-name, global-name> tuples constitute the Eternal Root. These pairs are held in the form of links and not as small databases. This allows the user to see the object names as normal file names in their directories.

Because IK is a decentralized system, a globally unique tag is assigned to each node, which can then be combined with locally generated values to create globally unique identifiers. IK also uses the unique tag as a hint to the location where the object is most probably found. In IK, the storage server is where the objects are finally stored. The storage server can hence be used as a location hint to access an object in it. IK, therefore, assigns globally unique tags only to storage servers and requires the application's GRLs to request global names from the storage servers where the object will eventually be stored. If an object migrates from the default server, a forward pointer is left behind.

IK has location algorithms that are protocols between the application's GRL and the storage server that actually finds the object when the global name of the object is given to it.

Accessing objects via links

Morrison *et al.*[12] deal with persistent programming environments that allow software engineering environments to be completely supported within the persistent environment. Thus, each software environment component or activity can take advantage of the persistent system. According to the authors, persistent linkage that allows persistent objects to be included in the binding process can be used in combination with other properties like referential integrity and Strom typing to improve the construction and use of software engineering environments. The term refers to the reference to an object in the persistent environment. Once a link to the persistent object is established, the object remains accessible via the link as long as the link exists. This leads to the referential integrity of the links. The use of links with referential integrity increases the safety of the system. Instead of referring to the objects by some naming convention, anonymous links that have a close analogy with the i-node numbers in the UNIX operating system can be used. Another advantage of using links over names, according to the authors, lies in the fact that since access to the objects is independent of the naming scheme, any number of naming schemes, including zero, may be layered on top of the linking graphs for the convenience of the user. The user need not know the depth of the naming scheme. One disadvantage of the use of links over names is the reduction in flexibility when dealing with the persistent system. This is due to the fact that the decisions about which particular objects to access are made earlier.

The use of such links to persistent objects has a number of applications in the area of software engineering. It helps in simplifying the programming model, in version control, in configuration management, and in documentation. Hyper-programs and hyper-texts can be bound to the software applications due to links provided in such persistent systems.

POINTER SWIZZLING

Pointer swizzling is the conversion of database objects between

an external form of object identifier and an internal form of direct memory pointers. This concept can be used in most of the persistent environments. The main motivation behind pointer swizzling is to speed up manipulation of memory resident data. Swizzling is used in some object-oriented databases, persistent object stores and persistent and database programming language implementations.[7] Swizzling is also relevant to object stores and object servers, as applications using a store or server might benefit from converting objects from the store/server format to a faster in memory format.

Eliot and Moss[7] designed and performed experiments to evaluate the different schemes of pointer swizzling and to predict their performances. The Mneme object store was used as a platform for the performance evaluation. Swizzling generally saves time if enough computations are done with the swizzled objects. Therefore, if an application visits objects only once or only a few times, swizzling is unlikely to help. This is because swizzling involves conversions to and from memory. However, swizzling does not result in orders of magnitude impacts on performance. Hence, it is reasonable to choose a swizzling scheme based on factors other than performance. The cost of swizzling is found to go up when the object size is increased and when there are a number of pointers to be considered and swizzled. The collection of objects that are used by various applications running on the persistent system determines the extent to which swizzling can be beneficial. The decision of whether swizzling is desirable can hence be made based on the collection of objects.

The pointer swizzling techniques can be classified into three main areas on the basis of how swizzling is actually achieved. The three techniques are as follows:

- *In-place and copy swizzling.* In this scheme the objects in which the pointers are swizzled remain on their pages (in-place) on which they are resident on secondary storage or are copied (copy) into a separate object buffer. In-place swizzling requires that all the swizzled pointers contained in an entire page be restored or un-swizzled when the page is written back to secondary storage. Under copy swizzling, the objects are moved or copied from their pages into the object buffer and pointers are swizzled only in these copies. Copy swizzling, hence, is a better choice over in-place swizzling.

- *Eager and lazy swizzling.* Eager swizzling guarantees that all the pointers that are in main memory are swizzled. On the other hand, lazy swizzling swizzles pointers only on demand. In lazy swizzling, therefore, a pointer is not swizzled until the object it refers to is accessed via this particular pointer. No pointer is swizzled in vain in the case of lazy swizzling. On the other hand, lazy swizzling has to handle swizzled and un-swizzled pointers at runtime. Hence, if a low percentage of pointers that are loaded into main memory are actually used to access the object they reference, then lazy swizzling is a better choice over eager swizzling.

- *Direct and Indirect swizzling.* When direct pointer swizzling is used, the swizzled attribute contains a direct pointer to the referenced in-memory object. Under indirect swizzling there is one indirection in which the attribute contains the pointer to a descriptor, which then contains the pointer to the referenced object.

Lazy swizzling costs more than eager swizzling and this was also seen through experiments performed on the Mneme object store.[7] Similarly, copy-swizzling costs more than in-place swizzling, for objects of the same size, when the objects are updated. But, this difference is not too much. Hence, the authors conclude that when there is adequate main memory, copy swizzling is a better choice over in-place swizzling. This is because copy swizzling allows much more general transformations between disks and memory formats.

The costs or benefits of swizzling depend to a large extent on an application's use of the objects. In this regard, the size of the objects is important. A particular persistent system may use swizzling, or it may not. This may be decided based on additional issues beyond performance issues.

The details of pointer swizzling techniques and the system tables that need to be maintained to perform the pointer swizzling have been studied in Object Database Management Systems (ODBMSs) by Vadaparthy.[17] In the case of ODBMSs data is retrieved from the persistent store. This data is processed by using some operation defined in the host language, which could be C++, and the data is stored back into the persistent store.[18] Hence, even in ODBMSs every kind of processing requires translation back and forth. Such translations can be time-consuming if the frequency of application of the operation is high. Hence persistent pointers are used to a large extent in ODBMSs. To improve the performance of main memory accesses, a similar concept of pointer swizzling can be used in such systems.

EVALUATION

In this section we evaluate the different schemes for persistent object store organization and accessing the persistent objects that have been discussed in this article. As we have already mentioned, there are a number of persistent environments based on the underlying architectures that are used to implement the persistent systems. The persistent environment can be centralized or distributed. The treatment of persistent systems is, therefore, different, and is based on the environment that is present. In this article we have mainly dealt with the persistent environments of the O2 system, Arjuna, IK system, and a few other persistent environments. The first three are object-oriented platforms supporting a distributed environment and persistent applications.

It is important in a persistent environment that a persistent object or logically grouped objects in the system can be retrieved easily and at a fast rate from the persistent store. The persistent store organization, therefore, contributes to the performance of retrieving the objects from the persistent store. In this article we have emphasized the importance of clustering objects in the persistent store. This clustering can be physical clustering or logical clustering. Physical clustering groups objects that are instances of the same class together, whereas logical clustering generally

considers the inheritance characteristics of the class of which the objects are instances. In a persistent environment objects can refer to other objects leading to the concept of complex objects. In such a situation, the idea of logical clustering of the objects on the object store makes better sense than the concept of physical clustering of the objects.

With logical clustering it is fair to say that the performance of retrieving the complex objects from the object store will be faster. This is because the referring object and the referred objects will surely be on one page of the persistent store. It may happen that a cluster might not fit into a page. In the O2 system[2] when a placement tree leads to clusters whose average size is greater than the page size, such a tree is to be rejected. The best placement tree is then to be found from those remaining.

Whatever scheme the persistent store uses, the users should not need to worry about physical issues, i.e.,the clustering information needs to be transparent to the user. In this regard, object-oriented systems or databases that rely on physical object identifiers and use user hints for their clustering mechanisms, may lead to the difficult situation of the user having to deal with the complex clustering problems. The scheme that is used by Arjuna[6] enables users to create their own object stores to manage any classes that they may define. While this will give the user more flexibility to deal with the clustering mechanism directly, it may lead to a complex situation. Therefore, this is not a very good choice for the clustering mechanism.

Arjuna uses a combination of physical and logical clustering. In Arjuna, objects that are instances of the same class are maintained together. For complex objects, the entire logical group of objects is implemented as a collection of individual object stores, one for each class of objects. With this scheme, we cannot ensure that a logical group of objects are as close together as possible to minimize the disk accesses. Hence, the retrieval time or the performance of the retrieval may not be the best. More than one disk access may be required in order to get the logically grouped objects into the main memory. If more than one page is required in such a case, then there could be the possibility of a higher number of page faults.

The O2 system,[19] on the other hand, uses a clustering mechanism that relies on the inheritance and the structural information given by the object types. Instead of clustering all the components of the object together, grouping strategies are defined. Placement trees are obtained which take the operations performed in the database into account. This will ensure that the logically grouped objects are as close to each other as possible when the best placement tree is chosen.

The IK object-oriented[16] persistent system uses a clustering strategy somewhat similar to that found in the O2 system. In this system, objects are assigned to clusters and each cluster has a singly known object called the head object that is similar to the referencing object of the O2 system. Using the head object, the referenced objects can also be retrieved. Hence, the referenced objects do not need a unique identifier for their retrieval from the object store. The clusters are created by using the KOT and therefore it takes some time to create the clusters initially.

The naming mechanisms used in Arjuna and the IK systems are somewhat similar in that they use symbolic names for the objects to be accessed. These symbolic names are mapped to the system level names to retrieve the objects from the persistent store. Both the systems are distributed in nature. Arjuna uses a three-part naming mechanism that forms the fully quantified name of the object in the distributed network. The user may just specify the partially quantified name to the system to retrieve an object. The Name Server on Arjuna maps the partially quantified name of the object to its equivalent, fully quantified name. The user is not concerned with such a conversion. This gives flexibility to the user and he need not know on which node the object is actually located.

The use of links, with referential integrity in place of names, to access the objects on the persistent store in persistent systems can be taken advantage of when we are dealing with software engineering applications. This scheme can be used in programming environments. The concept of pointer swizzling that improves the performance of accessing the persistent objects that are already in main memory has also been discussed in this article. The pointer swizzling schemes, besides being used for faster main memory accesses of the objects, can also be used for faster accesses of objects from the persistent store along with the use of clusters. Pointer swizzling can also be extended to various programming environments.

The inclusion of persistence in a programming language is to provide a total interactive programming environment. The first language to use the idea of persistence was probably APL. The addition of persistence to PS-Algol showed that the principle could be extended to a variety of languages that support a heap.[1] Some languages, like Pascal, have been extended with object-oriented features to provide a possible foundation for persistence. The idea of persistence was later adopted by Smalltalk.[5] Programming languages like C++ can be extended to support persistence as seen in object-oriented database systems.[20] Laurent Silverio[10] explains how it is possible to make polymorphic objects in C++ persistent. Shilling[15] presents basic algorithms for storing and retrieving persistent object instances, and explores the trade-offs between various mechanisms for providing incremental persistent objects in C++. There are a variety of ways to augment a language like C++ with persistence.[11] Park *et al.* present a technique called forced inheritance for providing orthogonal persistence in C++.[13] Hence, the area of persistent systems is gaining popularity not only in research areas, but also in commercial areas. Extending persistence to the present programming languages is a challenge and an interesting area for further study.

CONCLUSION

We have dealt with three of the important aspects of persistent systems in this article. The aspects are: persistent store organization, schemes to access persistent objects that are present in the persistent store, and pointer swizzling. Various clustering mechanisms pertaining to different persistent environments have been dealt with. The choice of logical clustering over physical clustering is a better one to improve the performance of accessing and retrieving persistent objects from the persistent store.

Various naming mechanisms and the use of links to access persistent objects have also been discussed. The use of symbolic names at the user level to access the objects is a great advantage because the user need not know how the persistent system names and stores the objects. The use of pointer swizzling to improve the performance of retrieval of objects already in main memory is an important concept. Concerning this, future work could involve the use of the pointer swizzling scheme in programming languages which have been extended with persistence. ■

References

1. Atkinson, M., *et al.* "An Approach to Persistent Programming," *The Computer Journal*, 26(4):360–365, 1983.
2. Benzaken V., and C. Delobel. "Enhancing Performance in a Persistent Object Store: Clustering Strategies in O2," *The Fourth International Workshop on Persistent Object Systems*, pp. 403–412, 1990.
3. Cahill, V., R. Balter, N. Harris, and R. Pina. *Commandos Distributed Application Platform*, Springer-Verlag, New York, 1993.
4. Chou, H., D. DeWitt, R. Katz, and A. Klug. "Design and Implementation of the Wisconsin Storage System," *Software-Practice and Experience*, 15(10), 1985.
5. Cockshott, W. "Persistent Objects in Turbo Pascal," *Journal of Object-Oriented Programming*, 6(2):68–73, May 1993.
6. Dixon, G., G. Parrington, S. Shrivastava, and S. Wheater. "The Treatment of Persistent Objects in Arjuna," *The Computer Journal*, 32(4):323–332, 1989.
7. Eliot, J., and B. Moss. "Working with Persistent Objects: To Swizzle or Not to Swizzle," *IEEE Transactions on Software Engineering*, 18(8):657–673, 1992.
8. Kato, K., and T. Masuda. "Persistent Caching: An Implementation Technique for Complex Objects with Object Identity," *IEEE Transactions on Software Engineering*, 18(7):631–645, Jul. 1992.
9. Kemper, A., and G. Moerkotte. *Object Oriented Database Management*, Prentice Hall, Englewood Cliffs, NJ, 1994.
10. Laurent, P., and V. Silverio. "Persistence in C++," *Journal of Object-Oriented Programming*, 10(4):41–46, 1993.
11. Loomis, M. "Making Objects Persistent," *Journal of Object-Oriented Programming*, 6(6):25–28, Oct. 1993.
12. Morrison, R., *et al.* "Exploiting Persistent Linkage in Software Engineering Environment," *The Computer Journal*, 38(1):1–16, 1995.
13. Park, C., K. Whang, I. Song, and S. Navathe. "Forced Inheritance: A New Approach to Providing Orthogonal Persistence," *Journal of Object-Oriented Programming*, 9(1):65–71, Mar. 1996.
14. Shannon, K., and R. Snodgrass. "Semantic Clustering," *The Fourth International Workshop on Persistent Object Systems*, pp. 389–402, 1990.
15. Shilling, J. "How to Roll Your Own Persistent Objects in C++," *Journal of Object-Oriented Programming*, 7(4):25–32, 1994.
16. Sousa, P., A. Zuquete, N. Neves, and J. Marques. "Orthogonal Persistence in a Heterogeneous Distributed Object-Oriented Environment," *The Computer Journal*, 37(6):531–541, 1994.
17. Vadaparthy, K. "Pointer Swizzling at Page Fault Time," *Journal of Object-Oriented Programming*, 8(7):12–20, Nov. 1995.
18. Vadaparty, K. "Persistent Pointers," *Journal of Object-Oriented Programming*, 8(4):14–18, Jul. 1995.
19. Velez, F., *et al.* "Implementing the O2 Object Manager: Some Lessons," *The Fourth International Workshop on Persistent Object Systems*, pp. 131–138, 1990.
20. Vemulaparti, M., D. Sriram, and A. Gupta. "Incremental Loading in the Persistent C++ Language," *Journal of Object-Oriented Programming*, 8(4):34–42, 1995.

CLIENT/SERVER

continued from page 19

RDB servers" today. It does not mean that the market does not need an ORDB that can deliver solutions to the issues of 1) managing complex structured data in a highly scalable way, 2) managing arbitrary data types in a highly scalable way, 3) directly supporting object-oriented programming, or 4) an RDB that can deliver solutions to the issues of unifying heterogeneous databases. Unfortunately, however, today's ORDB products are not ready for prime time.

One major technical challenge that ORDB vendors must meet is to make their servers highly scalable. The single server-process architecture, and a two-tier client-server architecture that ORDBs support today is untenable. The servers must be made to run on parallel computers (at least the Symmetric MultiProcessors), to support at least a three-tier client-server architecture, and to manage terabytes of data and at least hundreds of simultaneous users. Another key technical challenge for ORDB vendors is to interface their ORDB servers with popular rapid application development (RAD) tools and popular applications. These are the two areas where ORDBs fall sadly behind RDBs. Either ORDB vendors shore up these deficiencies, or giant RDB vendors will slowly but surely upgrade their servers to real ORDB servers, and upgrade their RAD tools and applications to work with real ORDBs. Regardless of who "gets there" first, ORDB is here to stay. Further, once ORDBs deliver the four key benefits outlined herein, the existence of OODBs becomes precarious indeed. ■