A Survey of Architectural Features of Contemporary Object Storage Systems*

Jung-Ho Ahn, Sang-Won Lee, Ha-Joo Song and Hyoung-Joo Kim

Department of Computer Engineering, Seoul National University,
Shilim-Dong Gwanak-Gu, Seoul 151-742, KOREA
Tel: +82 2 880 5379, Fax: +82 2 871 6945,
EMail: {jhahn,swlee,hjsong,hjk}@oopsla.snu.ac.kr,
URL: http://wwwoopsla.snu.ac.kr

Abstract

In recent years, many object-oriented database systems have been developed and have become widely accepted in the DBMS commercial market. An efficient object manager, which is a middle layer on top of a storage system, is essential to ensure reasonable performance of object-oriented database systems, since a traditional record-based storage system is too simple to provide object abstraction.

In this paper, we first explore the current state-of-the-art of object storage systems – object managers in combination with storage systems – by focusing on three main architectural issues: client-server architecture, object identifier representation, and the object access method. Then, we identify the dependencies between these factors and propose a general guideline for the design and implementation of an ideal object storage system that provides transparency, high performance, and flexibility. In addition, we briefly discuss transaction processing, object clustering, and object versioning that are closely related to improving the performance and the modeling power of object storage systems.

Key words: Object manager, Storage system, Object-oriented database system, Survey

* This work was partially supported by Ministry of Education through Inter-University Semiconductor Research Center (ISRC 96-E-2026) in Seoul National University and by the Ministry of Trade, Industry, and Energy of KOREA under project Electro-21.

1 Introduction

The advantages of object-oriented databases in terms of a rich data model for next-generation database applications, such as CAD/CAM/CASE, AI expert shells, and multimedia office information systems, have become widely recognized [6, 14]. Due to the complexity of data management in such applications, key issues are performance and the requirement for a flexible and transparent object management environment. To meet these stringent requirements, there have been numerous works concerning high-performance object management [4, 6, 10, 14, 16, 17, 18, 24, 30, 41, 63].

Contemporary relational database systems consist of two main modules: a query processor and a storage system. A query processor returns the result of a given query by translating it into a series of internal storage system calls. The low-level storage system provides data persistency and transaction management with full control of physical devices. In object-oriented database management systems (OODBMSs), however, it is no longer adequate for upper layers, such as a query processor, to call a low-level storage system directly. This is because the upper layers of an OODBMS should be adapted to the rich and extensible nature of the object-oriented data model directly, while a traditional relational storage system supports only record-oriented data abstraction. That is, upper layers (if built directly on top of the relational storage system) would have to implement object abstraction, resulting in poorer performance due to increased complexity [6].

To overcome this problem, most OODBMSs employ a middle layer, which is called an *object manager*, on top of the storage system. The objective of an object manager is to reduce the impedance mismatch between upper layers (e.g., object query processor) and lower layers (e.g., storage system) by implementing object abstraction using the facilities of the underlying storage system. Along this line, an efficient object manager is essential to ensure reasonable performance of OODBMSs.

We summarize the basic functionalities of an object manager as follows [6]:

(1) to generate object identifiers
(2) to create and delete persistent objects
(3) to support object access method
(4) to support object naming service

Besides the above features, an object manager is also involved in method binding, object versioning and object clustering.

The object manager, together with the storage system, is referred to as an object storage system. In this survey, we explore the current state-of-the-

Table 1
Products, Vendor, and References

Product	Producer/Vendor (References)
EXODUS [1]	Univ. of Wisconsin-Madison [19, 20, 50, 62]
Itasca (V 2.3)	IBEX Object Systems, Inc. [26, 27]
O_2 (V 5.0)	O_2 Technology [6, 54, 55]
Objectivity/DB (V 5.0)	Objectivity, Inc. [44, 45]
ObjectStore (V 5.0)	Object Design, Inc. [37, 42, 43]
Ode<EOS> (V 4.2)	AT&T Bell Laboratories [1, 2, 5, 9]
Ontos DB	Ontos, Inc. [31, 46]
Versant (V 5.0)	Versant Object Technology Corp. [58, 59]

art of object storage systems by focusing on three main architectural issues: client-server architecture, object identifier representation, and object access method. In particular, we are interested in understanding the dependencies between these factors and drawing a general guideline directing the design and implementation of high-performance object storage systems. In this paper, we only consider qualitative aspects of these architectural issues, since the quantitative analysis is beyond the scope of the paper.

Table 1 lists several commercial products and research systems that we examine throughout the survey. On some issues, we also examine a few other systems including GemStone [39] and QuickStore [63].

The remainder of the paper is organized as follows. Sections 2 through 4 describe the main architectural issues that must be addressed in the implementation of object storage systems. First, in section 2, we discuss common system architectures for building object storage systems. In section 3, we compare alternatives for implementing object identifiers. In section 4, we review the mechanisms for accessing objects and discuss four topics in detail including object fault handling and pointer swizzling. In section 5, we briefly describe additional techniques of transaction management, object clustering, and version control, all of which are aimed at improving the performance and the modeling power of the system. After this review, in section 6, we analyze the dependencies and interrelationships among architectural factors and their effects on performance, flexibility, and transparency. Finally, the summary of our survey and some areas for future research are given in section 7.

[1] We target EPVM (V 2.0) built on top of the EXODUS storage system (V 3.0)

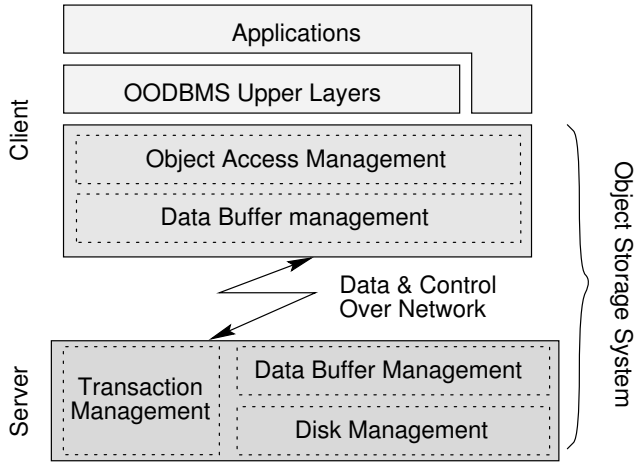


Fig. 1. An architectural abstraction of object storage systems

2 Client-Server Architecture

Early database systems usually ran on mainframes together with application programs. However, the rapidly increasing demands of managing distributed artifacts and the growth of computer hardware and software have brought us to the development of client-server computing [10]. Clients have replaced many functionalities that were previously performed by a centralized server. Therefore, the overall system performance has been enhanced by reducing the load on the server. Moreover, this architecture can fully utilize the advantage of high-performance workstations or PCs. Contemporary RDBMSs (relational database management systems) support a client-server environment that is typically based on *query-shipping* architecture where a server processes queries that are shipped from clients. In contrast to traditional database systems, OODBMSs usually ship data (namely *data-shipping*) from a server to clients so that clients can navigate the shipped data and perform query processing locally by themselves [13]. The reason is that OODBMS applications usually require a great deal of navigational data access and are computationally intensive, while those of RDBMS tend to depend heavily on sequential scans with little computing. As such, in OODBMS applications, caching the frequently used data in clients can provide faster data access [14].

Data-shipping architecture also provides good scalability. With query-shipping architecture, a server becomes the performance bottleneck as the number of clients increases. However, in data-shipping architecture many parts of database functions such as data caching, traversing, and even query processing, are moved to clients to exploit their additional computing power. This approach can provide good load balancing between clients and servers, and make the most of the ever-increasing power of workstation or PC clients.

Figure 1 illustrates the common architecture of object storage systems. The

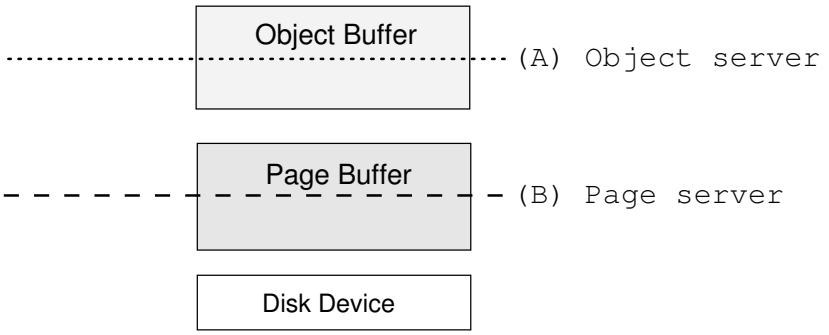


Fig. 2. The alternatives of buffer configuration

server provides low-level database functions such as disk management, buffer management, and transaction management. The object access management layer in the client provides applications with transparent access to the persistent data.

Both clients and servers maintain data buffers to keep recently accessed data so as to minimize network traffic and provide fast data access. Some object storage systems have object buffers for caching objects used by upper layers [33, 45]. This kind of buffer configuration, where an object buffer works on top of a page buffer, is called *dual-buffer architecture* (see section 4.4). Figure 2 shows possible client-server architectures from the viewpoint of buffer placement. The object server architecture results when a server ships individual objects to clients, and the page server architecture represents the approach where clients buffer pages that are brought from a server. Object server and page server architecture divide clients and server by the lines `(A)` and `(B)`, respectively. In page server architecture both the server and the client have a page buffer, but a client may or may not have an object buffer [2]. With the object server approach, a client has only an object buffer and a server has both a page buffer and an object buffer. However, a server may not have an object buffer when it ships objects extracted directly from a page buffer (e.g., Versant).

EXODUS, O_2, Objectivity/DB, ObjectStore, and Ode take page server architecture, while Itasca, Ontos, and Versant employ object server architecture [3].

Bancilhon et al. [6] evaluated these two representative data-shipping techniques with respect to performance issues. In that paper, they drew conclusions concerning shipping granularity as follows. The main advantages of object server architecture are the high space utilization of a client buffer, less sensitive to clustering, and fine-grain concurrency control and recovery.

[2] Many current page server systems do not have an object buffer, since they access objects directly in the page buffer to improve performance (see section 4.4).
[3] Ontos and Versant also have APIs (application programming interfaces) to ship a group of objects at a time.

However, too many interactions between clients and servers (network traffics) would induce a significant performance degradation. While the page server architecture avoids network overhead by shipping a set of objects rather than one object, the actual performance is highly dependent on the effectiveness of object clustering.

Carey et al. [11] also studied the granularity issues in data shipping architectures and presented an 'adaptive locking' policy which allows the system to lock objects if there is a page-level conflict. O_2 uses this protocol.

3 Object Identifier

Object identity (OID) is one of the most important features in the object-oriented data model. In object-oriented databases, each object has its own unique identifier which remains invariant, independent of the object's value and structure, throughout the lifetime of the database [8, 13, 18, 33]. More importantly, OID allows an object to directly reference another object (possibly itself) in the database. To guarantee referential integrity between objects, most systems do not re-cycle the OIDs of deleted objects.

In object-oriented databases, OID provides efficient navigational access by direct representation of the relationship between objects, while relational databases represent relationship by values and perform costly joins to traverse along the relationship between records. This is the reason why OODBMS can provide better performance in applications such as CAD/CAM and CASE that involve a large number of complex objects.

Due to heavy use of OID in object-oriented databases, its representation has considerable impact on the system performance. The number of disk accesses required to retrieve an object through its OID depends mainly on the OID representation. In this section, we deal with this issue after introducing the taxonomy of persistent OID representations. Dependency with other architectural issues will be discussed in section 6.

Khoshafian and Copeland [32] introduced nine different OID representations that come from programming languages and databases, and compared them using a taxonomy based on data and location independencies. However, as pointed out in Khoshafian and Copeland [32], some of them are not appropriate for a persistent environment since persistent applications require a strong representation of OID that needs to survive after a program terminates. For this reason, two kinds of OID representations are mainly used by object-oriented database systems: *physical OID* and *logical OID*. These correspond to the terms, *structured id* and *surrogate* in Khoshafian and Copeland

[32], respectively.

A physical OID encodes the permanent address of the object referred to by itself. This characteristic makes it generally possible for object storage systems to obtain an object from a disk in a single disk access. EXODUS, O_2 [4], Objectivity/DB, ObjectStore [5], and Ode use the physical OID scheme. On the negative side, this technique lacks location independency [32]. That is, objects cannot simply be moved to another location in a database. Although objects can be moved around by indirections, such as forward marks [6], database reorganization is very difficult since it introduces numerous indirections which may degrade access performance.

In contrast to physical OID, a logical OID is generated by the object storage system independently of the physical address of an object. Thus, this representation allows flexible object movements and replications. Ontos and Versant follow a logical OID scheme. In addition to uniqueness, the logical OID can include a type identifier to access type information quickly. Itasca uses this kind of OID – *typed logical OID*.

However, logical OID schemes may degrade the overall performance of the system, since information on mapping between logical OIDs and their physical addresses must be maintained to locate objects. Many systems employ hash or B-tree structures to speed up object access [18]. Itasca and Versant use hash-based mapping techniques. In particular, Itasca maintains a hash index for each class separately. On the other hand, GemStone [39] uses B-tree index for mapping OID to a physical address. To our knowledge, there are only a few works on the performance of OID mapping techniques. One of these works is that of Eickler et al. [18] who evaluated the performance of three logical OID mapping techniques, including hash, B-tree, and a hybrid technique.

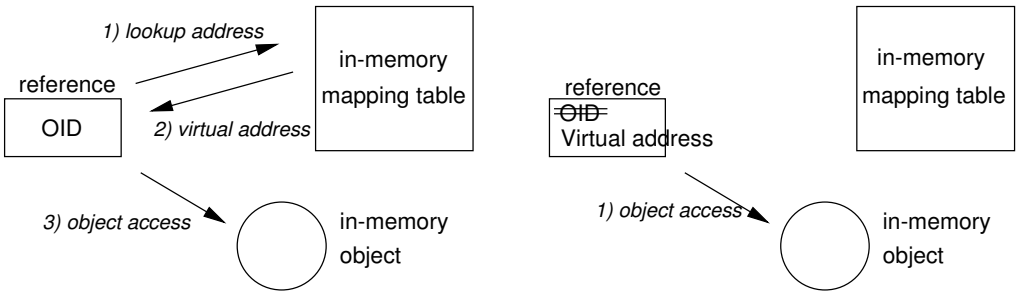
4 Object Access

All persistent objects need to be brought into main memory, since we cannot access the objects directly on disks. That is, object residency should be checked on every access, and then the object must be fetched into main memory if necessary. Such a residency check and handling mechanism is called *object fault handling* [24].

Object storage systems usually maintain mapping tables to locate objects

[4] O_2 can also export objects with a universal identifier that can be used to retrieve the object at any time.

[5] ObjectStore always keeps OID fields in a swizzled form to speed up object access (see section 4.2).



(a) without swizzling (b) with swizzling

Fig. 3. The scenario of accessing in-memory objects

cached in main memory, as shown in Figure 3(a), and object residency is checked during the OID translation into the corresponding virtual address. However, it is a performance penalty to check the residency and compute the in-memory address on every access. To solve this problem, many object storage systems replace in-memory OID references with virtual addresses. Figure 3(b) shows this concept that is known as *pointer swizzling* [41]. Pointer swizzling can improve the performance of object access by skipping the lookup-table search, particularly in CPU-intensive applications. It can also give transparent access to persistent objects just as for transient objects.

Efficient management of in-memory objects is also required, since object-oriented database applications have a strong tendency to cache a large number of objects and perform extensive computations on them.

As such, major issues of accessing persistent objects are as follows:

- Fault handling: How can nonresident object access be detected and handled?
- Pointer swizzling: How many pointers should be swizzled at a time? And how can swizzling techniques be combined with residency checking?
- Access interface: What kind of interface is provided to access objects?
- Object management: How and where are resident objects managed?

The mechanism and performance of accessing persistent objects depend primarily on the above four topics, and these issues are closely interrelated. In this section, we discuss each of the four issues in detail and their architectural dependencies.

4.1 Object Fault Handling

There are two issues related to the object fault handling problem [24]: one is how to detect access of a nonresident object and the other is how many objects

need to be fetched per object fault. In-memory hash tables are commonly used to detect nonresident object access: for example, O_2, Objectivity/DB, Ode, and Versant. That is, the residency of an object can be checked during the in-memory address calculation via hash table lookup.

If the object referenced is not memory-resident, the system has to bring it into the buffer. The problem here is how many persistent objects should be brought into main memory per object fault. In respect to this design choice, we can classify object fetch policies into three [24, 41, 61]: 1) *object-at-a-time scheme*, which makes only the missed object resident, 2) *page-at-a-time scheme*, where all objects from a page or a segment [6] are extracted together upon the first fetching of an object from the page or the segment, and 3) *closure-at-a-time scheme*, where all objects linked with the faulted one are fetched recursively on every object miss.

The object-at-a-time fetch scheme ensures the high space utilization of a client buffer and it is less sensitive to clustering. However, too many object faults would degrade the performance of object access significantly.

The page-at-a-time fetch scheme allows good performance of object access by improving the buffer hit ratio, and it can be highly profitable in multi-client environments, since object hits save work load on the server [16, 25, 29]. Many systems including O_2, Objectivity/DB, and Ode fetch all objects from a page eagerly upon the first miss of an object in the page [7]. However, when databases are clustered poorly or object access patterns are not incorporated with clustering, this policy may lead to many page faults as well as unnecessary copy overhead [16, 25, 47]. Thus, this approach may induce significant performance degradation although it can be of benefit in small and well clustered databases.

The closure-at-a-time fetch scheme makes residency detection unnecessary by fetching, in advance, all accessible persistent objects linked with an entry point (see section 4.2). This scheme guarantees extremely good performance for retrieving complex objects, if they are clustered well, and the data sets are small. However, this policy may cause the fetching of more unnecessary objects than the page-at-a-time policy if few of the linked objects are actually followed. It may also induce significant overhead to pre-fetch the transitive closure of a data set when the underlying object storage system has no idea about object semantics like class or relationship [57]. Moreover, this scheme cannot predict general object access patterns that might result from invoking methods [17].

[6] A segment can be a physical set of pages or a logical collection of objects.
[7] These systems usually keep objects in page buffers without copying them.

Pointer swizzling is based on the idea that improved performance of object access can pay off the swizzling (and unswizzling) cost [41, 62]. Although many approaches for swizzling have been proposed, their techniques can be categorized along two dimensions: how many pointers are swizzled at a time? and how does a swizzled pointer maintain object residency?

The first dimension classifies swizzling techniques into the following four approaches according to how aggressively they swizzle pointers [30, 41, 61]: *closure-at-a-time*, *page-at-a-time*, *object-at-a-time*, and *pointer-at-a-time*. In the closure-at-a-time swizzling scheme, the system recursively swizzles all of the pointers in the transitive closure of the referenced objects. Page-at-a-time and object-at-a-time swizzling schemes perform swizzling on all pointers within a page and an object at a time, respectively. Generally, the above three approaches, which swizzle pointers in advance, are considered to be *eager swizzling schemes* [8]. As pointer swizzling is performed more eagerly, it can cause more unnecessary swizzlings, and therefore possibly more unnecessary object fetches.

In contrast to these eager approaches, the pointer-at-a-time swizzling scheme, which is called a *lazy swizzling scheme*, waits until a pointer is actually used, and then swizzles only one pointer at a time. However, this policy increases run time overhead since every pointer access should determine whether the pointer is swizzled or not.

The pointer-at-a-time swizzling scheme can be further divided into two swizzling schemes: *upon-dereference* and *upon-discovery* [62]. The pointer-at-a-time upon-dereference scheme defers pointer swizzling until the pointer is actually dereferenced, thus avoiding any unnecessary swizzling. This policy may, however, leave pointers between persistent objects non-swizzled, since pointer fields of objects are often copied into temporary variables. Thus, the same pointer may have to be swizzled several times. On the other hand, the upon-discovery scheme swizzles a pointer as soon as its location is discovered by pointer operations such as comparison, assignment, and dereference. Consequently, this policy can solve the repetitive swizzling problem associated with the upon-dereference scheme although it may swizzle some pointers that will never be dereferenced. Figure 4 summarizes the classification of pointer swizzling techniques dependent on the first dimension.

A pointer swizzling technique should also provide a mechanism for detecting access to nonresident objects, since access through swizzled pointers does not involve in-memory hash table lookup. Along the second dimension – how to

[8] Moss [41] considers only the closure-at-a-time scheme to be an eager technique.

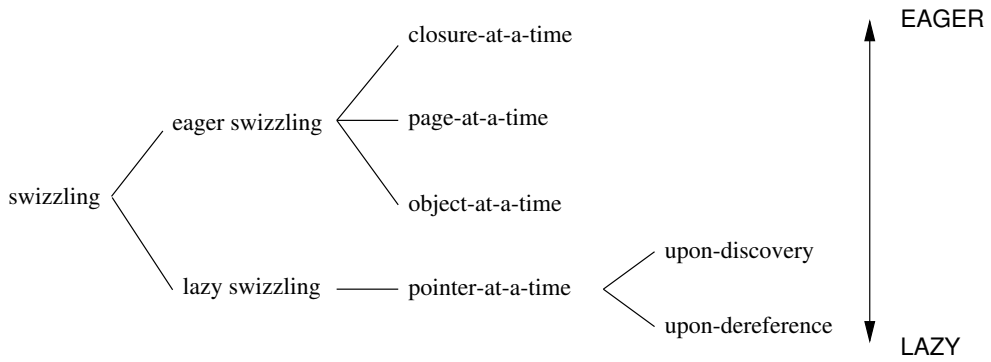


Fig. 4. Pointer swizzling techniques (depending on how many pointers to swizzle at a time)

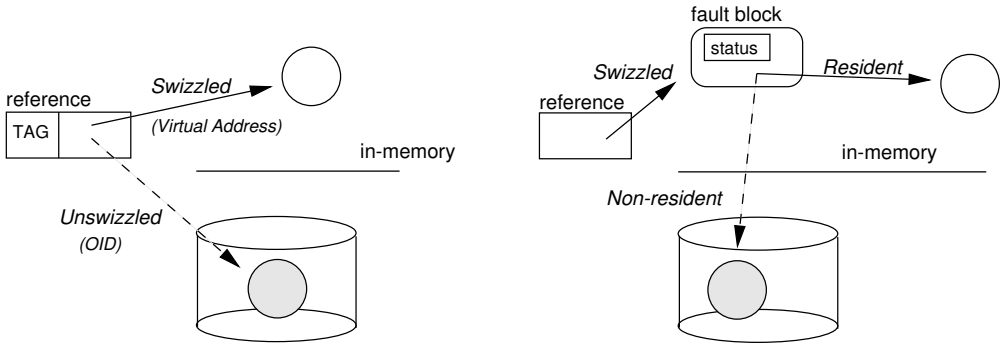
mark object residency – we can classify pointer swizzling policies into three [24, 41]:

- *No marking*
- *Edge marking*
- *Node marking*

First, a no-marking policy guarantees that applications meet only swizzled pointers that always reference resident objects, avoiding swizzling and residency checks entirely. However, this scheme requires closure-at-a-time fetch and swizzling policies that fetch all objects that may possibly be accessed and swizzle all pointers in them in advance.

Second, the edge-marking scheme ensures that swizzled pointers always point to resident objects, while nonresident objects are referenced through non-swizzled pointers [24, 41]. With an edge-marking scheme, pointers should be checked to see if they are swizzled on every pointer access, since the program can see both swizzled and non-swizzled pointers. This swizzling check is done by tagging references, as shown in Figure 5(a).

While an edge-marking scheme caches object residency in edges, that is, pointers, a node-marking scheme marks residency in nodes referenced by pointers [24]. Thus, this policy detects access to a nonresident object by checking the state of the node, and this scheme avoids any swizzling check by ensuring that all in-memory pointers are always swizzled. According to how the status of nodes is marked, node marking scheme can be further classified into *fault block method* and *memory protection method* [24]. In the fault block method, swizzled pointers contain the addresses of intermediate blocks, which are called fault blocks, instead of the addresses of objects. Fault blocks keep the residency status of objects and cache their virtual addresses if resident. As shown in Figure 5(b), objects are always accessed indirectly via fault blocks, and thus the replacement of unused objects is easier than it is with the edge-marking method. Kemper and Kossmann [30] refer to edge marking and the fault block



(a) Edge marking (Direct swizzling)

(b) Fault block (Indirect swizzling)

Fig. 5. Pointer swizzling techniques (depending on how object residency is marked)

method as direct and indirect swizzling, respectively.

The memory protection method marks the residency of objects by protecting the virtual memory spaces where the objects are located. The basic idea of this scheme is as follows. If the referenced object is not yet resident, the pointer is swizzled to point to a reserved address space where the object will be loaded later, and the reserved space is protected to detect access to this area [64]. An underlying operating system and hardware are responsible for trapping illegal access to the protected area and forwarding control to the object storage system. Object fault handling is achieved by fetching the missed object into the reserved space and then unprotecting the area.

Because protecting operations can normally be performed over multiples of the operating system's virtual memory page size, the memory protection method requires page-at-a-time object fetch and pointer swizzling schemes. This approach provides transparent and efficient access to persistent objects in the same manner as to transient objects, since access to a nonresident object is detected by an underlying hardware without any intervention by residency or swizzling check. However, the main advantage of this scheme may be lost due to the high fault cost, including that of reserving and protecting virtual address spaces and trapping page faults [63]. Also, the memory protection method imposes a stringent limit on the number of objects accessed in one transaction to the size of virtual address space, and address space reservation further reduces the memory space that is actually used. Moreover, illegal pointer operations can cause serious object corruption (see section 4.3).

There are several prototypes and commercial OODBMSs that employ memory protection methods: for example, Cricket [52], Texas [53], QuickStore [63], and ObjectStore. While object identifiers in Cricket become virtual memory addresses, Texas scans and swizzles all of the pointers in a page whenever

the page is fetched by a fault handling to support large address spaces. As described above, virtual memory spaces are reserved and protected to swizzle the pointers that reference nonresident objects. On the other hand, ObjectStore and QuickStore try to avoid heavy swizzling work by storing pointers on disk in swizzled form, that is, as virtual memory addresses [63]. In order to make stored virtual addresses valid throughout the lifetime of a database, the system always has to map disk pages at the same locations where the pages were the last time. However, several disk pages may be associated with the same virtual address due to the limitation of virtual address space. When the virtual address space (where the fault page was last mapped) is already allocated to another page, the faulted one has to be assigned a new virtual address space, not yet allocated. This re-mapping process requires additional swizzling work: all of the pointers that reference objects on the re-mapped page have to be updated to point the new locations. This is done incrementally by examining whether the current memory mapping does not violate the last mapping associated with the page on every page fault. As the size of the database increases, the probability of virtual address conflicts among disk pages increases since the same virtual address should be assigned to several disk pages.

Along another dimension, swizzling schemes may be classified into *in-place swizzling* and *copy swizzling* depending on whether or not objects are copied at the time of swizzling [41]. An in-place swizzling policy swizzles the pointer fields of objects in the page buffer directly, while a copy-swizzling scheme makes swizzled copies of objects in the object buffer (or application heap). Exploiting the dual-buffer architecture, copy swizzling can save the cost of unswizzling – that is, restoring a virtual addresses into an OID – since only new and modified objects have to be unswizzled. In contrast to the copy-swizzling scheme, in-place swizzling can save copying cost and memory resources.

There is a strong interdependence among the swizzling scheme, the architecture of an object storage system and the interface language. As mentioned earlier, fault handling policy – detection of nonresident object access and fetching missed objects – is also closely related to the swizzling scheme. Figure 6 shows the dependency among these schemes, and the approaches taken by several OODBMSs.

The first bar is the object fetch scheme. The second and the third bars show the previous two dimensions of our classification of the swizzling scheme, swizzling time and the swizzling (and residency) check method, respectively. As shown in Figure 6, there are three major approaches to accessing objects: no-swizzling, software-based swizzling, and hardware-based schemes. While

[9] EXODUS takes an edge-marking policy but a swizzled pointer points to the object indirectly, via a user descriptor.

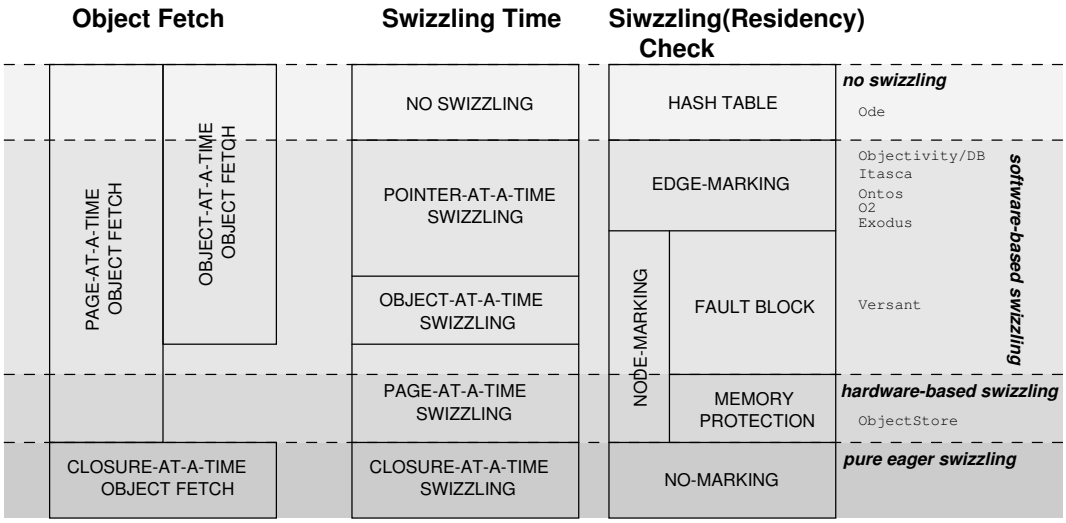


Fig. 6. Dependency of object fault handling and swizzling scheme [9]

hardware-based architecture, which uses the memory protection method, has to follow the page-at-a-time object fetch and swizzling schemes, software-based architectures – all swizzling techniques except the memory protection based method – have several combinations of object fetch and swizzling schemes. However, object-at-a-time or page-at-a-time swizzling schemes cannot be incorporated with edge marking, since they would then become identical to a closure-at-a-time policy. The figure also shows that no-swizzling and pointer-at-a-time swizzling schemes have no dependency with object fetch policy.

There have been numerous early studies on pointer swizzling, which is a major issue for high-performance object management [24, 30, 41, 62, 64]. Hosking and Moss [24], Moss [41] and White and DeWitt [62] made observations mainly about the various software-based implementations of pointer swizzling, and examined the performance of these alternatives. Kemper and Kossmann [30] argued that there is no clear winner in swizzling schemes and presented an *adaptable pointer swizzling policy* that employs several swizzling schemes adaptively, according to an application profile. Wilson and Kakkad [64] investigated techniques of hardware-based swizzling approaches, and Vaughan and Dearle [57] suggested a hybrid memory protection method that uses an object-at-a-time swizzling scheme instead of the page-at-a-time scheme. However, these earlier works have hardly considered the method and cost of unswizzling.

4.3 Object Access Interface

The object access interface is highly dependent on the object fault handling policy and swizzling technique. Access interfaces to an object storage sys-

tem [10] are classified into the following three [41, 61]:

- *Direct pointer interface*
- *Indirect pointer interface*
- *Call interface*

Direct pointer interface, which is usually provided by memory protection based systems such as QuickStore and ObjectStore, allows users to access persistent objects by normal virtual memory pointers. The advantage of direct pointer interface is that it is transparent and gives efficient access to persistent objects, just as for transient objects [63]. A disadvantage of this interface is that it can make the system unsafe by exposing actual in-memory addresses of objects. That is, illegal pointer operations may cause database corruption. The situation would be worse when clients share objects in the page buffer.

Indirect pointer interface gives object handlers instead of virtual addresses, and persistent objects are accessed by methods defined on the object handler. The dereference operation on the indirect pointer transparently performs object fault handling and pointer swizzling, and returns the object addressed by the pointer. C++ binding usually provides a smart pointer (which works, at least superficially, in the same way as a normal C++ pointer) as an object handler. However, the indirect pointer interface does not allow libraries, developed for transient objects, to be applied on persistent objects without change or re-compilation. Object access via indirect pointers also has a performance disadvantage, since every dereference operation incurs overheads such as a swizzling check or residency check. However the overhead of indirection becomes relatively smaller with increasing work on objects per access.

However, indirect pointers locate objects more safely than the direct pointers, and can be incorporated with various schemes of fetching, swizzling, and in-memory object management. Most OODBMSs that follow ODMG C++ binding guideline [12] support the indirect pointer interface: for example, O_2, Objectivity/DB, Ode, Ontos, and Versant.

With the call approach, objects are accessed by calling an object storage system. The call interface can be broken into several alternatives depending on what kind of access path is given. The first type is a naive call interface, which requires users to call the object storage system for every object manipulation. That is, persistent objects or even fields of objects are accessed by explicit function calls. The underlying system is allowed to detect all object accesses,

[10] Language bindings of OODBMSs are not always the same as the interfaces to object storage systems. For example, C++ and Smalltalk bindings of Versant are implemented on top of the object storage system that provides C API. Some of the examples we give here are the language bindings of OODBMSs, since we do not know the interface to the object storage systems exactly.

and thus guarantees the safety of object references. Although this interface makes persistent object management easy, it cannot avoid degradation of the performance of object access. This approach is used by EXODUS, where every object access in an E source code is converted to a call to EPVM [50]. Instead of calling the object storage system for each object access, it may be more efficient for function calls to return direct pointers. With this second type of call interface, users are allowed to use the direct pointers of in-memory objects freely until the pointers are explicitly released by function calls. However, users must take special care when dealing with the direct addresses of objects. Ontos and Versant support this limited form of direct pointer interface, which is not based on the memory protection method.

One of the issues for access interface is how the system should be informed of updates in order to request the right locks and to generate logs. Memory protection based systems can keep the track of object modifications transparently, taking advantage of the virtual memory hardware. This can be achieved by making attempts to write to a clean page incur a page fault.

As mentioned above, a naive call interface also allows a system to detect the modifications of objects transparently, since all the effects of updates are known to the underlying system. In all other cases, the user should mark dirty objects as modified explicitly, since the system cannot detect their updates automatically. For example, the `Ref` class in ODMG C++ binding provides a `mark_modified()` method to inform the underlying object storage system as to which objects are modified.

4.4 In-memory Object Management

The last issue for accessing persistent objects is where and how in-memory objects are to be managed. Depending on where persistent objects are brought into, we can classify two approaches of in-memory object management [13]. The first approach is to manage persistent objects directly in the page buffer pool of the underlying object storage system. To exploit the memory-protection scheme, QuickStore and ObjectStore map page buffer pools into virtual memory spaces and allow users to access objects there directly [63]. Also, most page server based systems such as O_2, Objectivity/DB and Ode keep in-memory objects in their page buffers. This policy can reduce copy overhead and keep the memory utilization high for well-clustered databases. However, these systems need to make object conversion such as pointer swizzling in place [13]. This policy also has the possibility of corrupting the system data, exposing the page buffer pool directly, and makes object resizing difficult.

When object instances are clustered poorly, it may not be desirable to keep

objects in page frames, since pure page-based buffering leads to inefficient space utilization [16, 29, 35]. To solve this problem, some OODBMSs (e.g., Itasca, Ontos, and Versant) are based on the dual-buffer architecture in which an object buffer functions on top of a page buffer, as mentioned above [16, 35]. This partitioned buffering provides good space utilization by filtering out useless objects from the page buffer and allows efficient object replacement and garbage collection. Also, the translation of objects between the disk representation and the in-memory format, including pointer swizzling/unswizzling, are easy and efficient [41]. However, the object buffer management has a number of complex and difficult problems that can prevent efficient object buffering. The object buffer should handle fragmentations as well as heavy memory allocations and deallocations, since it needs to manipulate a number of objects of various size. Moreover, the buffer consistency problem may make the object buffer management harder [16].

Whichever policy is used, unused objects should be displaced deliberately, with a notion of object access history for effective object buffering. However, it is not easy for a system to delimit a span of using an object, especially when the address (or pointer) of the object is accessed directly by a language like C++. So, adopting a replacement algorithm such as LRU for an object buffer might be problematic, if not impossible. Due to these difficulties, many systems including O_2, Ode, Objectivity/DB, and Versant keep objects in page or object buffers without object replacement until the transaction ends or the reference is definitely finished [11].

Previous works, attempting to increase the object buffer hit ratio, have investigated efficient object prefetch policies rather than efficient object buffer replacement algorithms, due to the difficulties involved with object buffering. Chang and Katz [14] proposed a policy that exploits high-level object semantics in terms of inheritance and structural relationship. Alternative approaches based on profiling or learning of object access patterns have been studied in Cheng and Hurson [16] and Palmer and Zdonik [47], and an object prefetch policy, which prefetches objects only from selected candidate pages without using any high-level object semantics, is proposed by Ahn and Kim [4].

[11] These systems also provide APIs to displace objects before the end of a transaction

5 Other Performance Sensitive Features

5.1 Transaction Processing

Transactions in new object-oriented applications may span long durations and involve human interactions, while traditional on-line transactions are short and flat [36]. Thus, object storage systems should employ new techniques of concurrency control and recovery. In this section, we only briefly describe new transaction facilities for object-oriented database systems, since this topic is beyond the scope of the paper.

Many OODBMSs support a *nested transaction* model and *check-out/check-in* policy. The nested transaction concept is the generalization of save-points used in traditional transaction models [23]. While save-points organize a transaction into a sequence of actions that can be rolled back individually, nested transactions form a hierarchy of transactions. That is, transactions can be nested recursively to an arbitrary depth [23, 40]. The top-level transaction controls the whole activity, while lower-level transactions, called sub-transactions, control each of the partial activities. An exception raised within a sub-transaction can be solved either by its parent transaction or by any surrounding transaction without aborting the whole activity. This mechanism is very useful for aborting a subset of works.

Cooperative work can be supported by the check-out/check-in mechanism, which is usually implemented by object versioning [12] [36]. The check-out/check-in model allows a user to extract one version of an object (or group of objects) from a group database into a private database. Objects in the private database can be manipulated by the owner without any intervention from other co-workers. When the user finishes the job on the objects, the user must check-in the updated objects into the group database so that other co-workers can share the results. Every check-out derives a new version of an object and thus, more than one user can work with the same object simultaneously – they all have different versions of the object. When a version is checked-in, it may be merged with another version to reconcile the differences between the two versions [43]. Itasca, Objectivity/DB, ObjectStore, and Versant provide check-out/check-in mechanisms with some variations.

In addition to the traditional concurrency control, new lock mechanisms such as *class locking*, *class hierarchy locking*, and *composite object locking* can be applied to OODBMSs [13, 33, 38]. Class lock can be further divided into *class definition lock* and *class instance lock*. The former means a lock on a class

[12] Check-out/check-in scheme can also be implemented by persistent locks: for example, Objectivity/DB and Versant.

definition itself, and the latter, a lock on all instances of a class. Class hierarchy lock, locks on all instances of a class and its sub-classes through the class hierarchy. Similarly, with composite object lock, all component objects are locked by locking only the root object; this mechanism requires that the object storage system understands the structural relationships between objects.

Notification is also an extension of traditional lock management [34]. This mechanism allows conflicts of object access, and transactions are notified of the conflict access immediately (*immediate notification*) or when one of them attempts to commit (*deferred notification*). A notification is transferred via e-mail or by triggering predefined actions on notified classes.

Caching in the data-shipping architecture gives performance enhancement, but also leads to cache coherence problems. There are two basic caching policies, caching within a transaction and caching between transactions, called *intra-transaction caching* and *inter-transaction caching*, respectively. With intra-transaction caching, all cached data must be invalidated when a transaction ends and a conventional two-phase locking protocol automatically keeps clients' caches consistent. That is, objects (or pages) which may have been cached by a previous transaction must be fetched and locked again whenever a new transaction begins, in order to keep consistency. Although the intra-transaction caching mechanism is easy to implement, it cannot benefit from inter-transaction reference locality [10, 21, 60]. On the contrary, the inter-transaction caching policy allows successive transactions to re-use data that have been cached by previous transactions. Two solutions for the cache consistency problem are widely used to support valid inter-transaction caching: *lock-validation* and *lock-callback*. With a lock-validation scheme, clients interact with a server to check the validity of cached data when it is first accessed in a transaction. This scheme reduces data transfers but still suffers from validation overhead. In contrast, the lock-callback approach keeps locks as well as data, even after a transaction terminates. When an object is accessed in a shared mode, the client does not need to either check the data validity or acquire a lock from the server. However, the client should get an exclusive lock to update an object, and the server should send an invalidation request to all clients for that object [43]. This type of reverse client-server communication makes the implementation of a lock-callback scheme difficult. Many commercial OODBMSs support inter-transaction caching. EXODUS, Itasca, Objectivity/DB, and Versant implement lock-validation schemes and, O_2 and ObjectStore follow the lock-callback approach.

Concerning recovery, conventional techniques such as *group logging* can also be applied to OODBMSs, since there is no significant difference from those of RDBMSs. However, data-shipping architecture enables improvement of the logging performance. For example, clients themselves can generate log records to reduce the burden on a server [48].

Table 2
Clustering Taxonomy

Policies	Immediate Clustering			Deferred Clustering		
	Class	Object	Set of Classes	Class	Object	Set of Classes
Products	Itasca Ode	EXODUS Objectivity/DB ObjectStore	Itasca Versant	-	-	O_2

5.2 Object Clustering

OODBMSs support object clustering, which places a semantically related set of objects on the same disk page or on adjacent pages to minimize disk I/O [14, 15, 31, 56]. On page server based systems, in particular, good clustering can dramatically enhance performance by minimizing interactions between clients and servers, and also by reducing the cost of locking and logging. Moreover, a well-clustered database provides good memory utilization, since most of the objects in a client buffer pool may be used.

We may classify clustering policies along two dimensions: when to cluster objects and how to specify clustering hints. According to the first dimension, the clustering strategy can be broken into *immediate clustering* and *deferred clustering*. In the immediate approach, the physical locations of objects are immediately determined at their creation time. Most existing OODBMSs take this approach. In contrast, the deferred approach delays the calculation of the physical locations of objects until a transaction commits. This strategy benefits from a global view of connections between objects, and consequently may achieve better clustering. O_2 follows the latter scheme [7].

Next, we can categorize object clustering policies into three according to the granularity of clustering hints. The first is the approach where all objects of a class are simply clustered together, as in RDBMS where all tuples of a relation are stored in a file [8]. That is, a class is the unit of clustering. Itasca and Ode follow this approach. The second policy allows a user to control the placement of an individual object at the time of its creation [14, 25]. In this policy, a new object is stored close to a 'near object' which is given as a clustering hint. The third is a more general scheme, which is supported by Itasca, O_2, and Versant. With this approach, all the instances of the set of classes can be clustered together according to clustering hints, such as a placement tree [7]. Table 2 summarizes the classification of object clustering policies and lists some examples.

Most available OODBMSs do not support dynamic re-clustering, which moves objects from their origins to other locations. In particular, it is hard to sup-

port dynamic re-clustering with a physical OID scheme because physical OID makes object movements difficult or useless, at least in terms of clustering [7]. In O_2 and Versant, though existing objects cannot be moved, a database administrator can change clustering hints at any time, something that would be effective only for newly created objects.

Many previous works have examined object clustering problems [14, 16, 22, 56]. Chang and Katz [14] described a clustering policy that exploits high-level object semantics in terms of inheritance and structural relationship. Cheng and Hurson [16] provided a dynamic re-clustering mechanism that takes into account object updates and multiple relationship. However, re-clustering is a difficult problem, as mentioned before. Gerlhof et al. [22] reported on an object database reorganization tool which monitors access statistics of applications, computes new object placement from the statistics, and finally restructures the database accordingly. Tsangaris and Naughton [56] investigated the performance of various clustering techniques for object-oriented databases, including depth-first search, breadth-first search, placement tree, and a stochastic algorithm.

5.3 Object Versioning

Design systems like CAD require data changes to be versioned so as to represent different design alternatives or evolutions [3, 28, 49, 51]. In addition, object versioning is used to implement a check-out/check-in model for long-duration transactions in cooperative work environments [27, 37, 54, 59].

Every versioned object – an object of which versions are created – has one root version, and all new versions are derived directly or indirectly from the root version. These 'derived-from' relationships between pairs of versions construct a version derivation hierarchy (VDH) [33]. Itasca and Ode allow only a tree-type of version graph, where more than one child version can be derived from a single parent. In O_2, Objectivity/DB, ObjectStore, and Versant, it is possible to derive a new version by merging two or more parent versions. However, a user must resolve the differences among the parents.

A versioned object may be referenced in two ways, by *static binding* and *dynamic binding*. In static binding, a specific version of the object is referenced statically. In contrast, dynamic binding determines the version referenced at run time. With a dynamic binding policy it is possible to maintain consistent reference to versioned objects automatically, without user intervention. All the systems that support object versioning, provide dynamic binding.

One of the most important concepts related to object versioning is version configuration [13]. A configuration is a specific set of versions that are con-

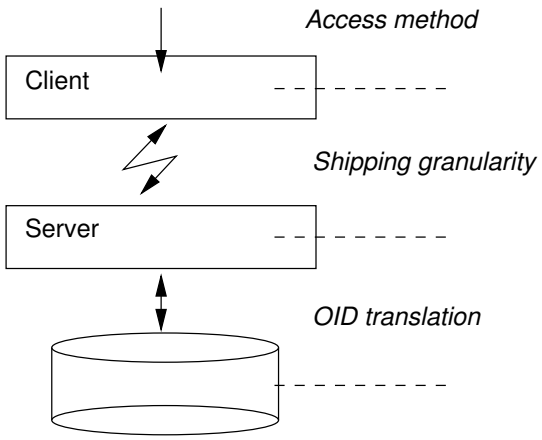


Fig. 7. Architectural issues

sistently associated with each other. O_2 and ObjectStore provide a simple and basic configuration management policy, but some other OODBMSs, such as Orion [33] and Ode, impose the burden of configuration management on the user just by providing object versioning and dynamic binding mechanism. Sciore [51] extended the EXTRA/EXCESS data model to allow users to specify configurations in a conceptual and declarative manner.

6 Dependencies among Architectural Aspects

In this section, we give a comparative review of architectural issues studied in the previous sections, and draw a general guideline for high-performance object storage systems.

The dependencies among the architectural factors can be illustrated by the scenario of passing a persistent object from disk to an application. Figure 7 shows how a persistent object is accessed through an OID. The object on the disk is first addressed, then brought into main memory. It is the representation of OID that affects the performance of this process, since the cost of locating and fetching objects is mainly dependent on the techniques for implementation of OID, as mentioned above. The next most important parameter is the shipping granularity, that is, how many objects should be transferred together. The granularity of data shipping actually implies the overall system architecture of an object storage system. Finally, the performance and transparency of access to objects cached in a client are determined by the access method, including the object fault handling mechanism and pointer swizzling policy.

As implied by Figure 7, each factor has close inter-relationships with adjacent factors – dependencies between access method and shipping granularity, and between shipping granularity and OID representation. It is worth emphasizing

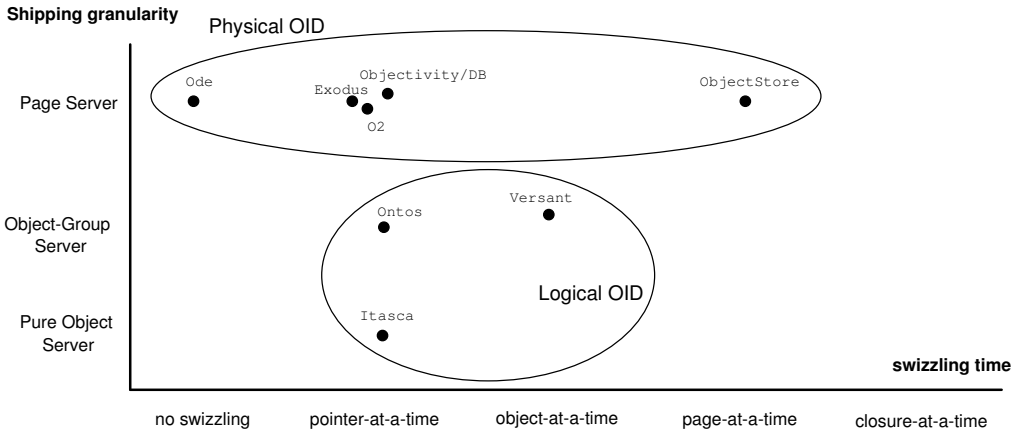


Fig. 8. Dependency of architectural issues

that these dependencies are not just interesting facts but are an important key to high-performance object storage systems. Our arguments can be confirmed by a review of existing products, as shown in Figure 8.

In Figure 8, we can easily see that there exists a strong dependency between OID representation and shipping granularity. That is, OODBMSs that employ physical OID take a page server approach, whereas logical OID drives to an object server approach. This tendency is explained as follows. Page server architecture is not appropriate for systems that use logical OID. With this environment, mapping information from logical OID to physical address should be shipped to clients and kept consistent. This makes object management, including creation, deletion, and movement, difficult and slow. On the contrary, object server architecture allows translation of OID to be performed at a server. Thus, the mapping information is maintained at the server only so that there is no problem of replication. Moreover, the average cost of OID translation can be comparable to that of physical OID, since most of the mapping information would be cached due to its hotness.

Considering the dependency between shipping granularity and the access method, closure-at-a-time and page-at-a-time swizzling schemes require an object server and a page server, respectively, due to their own architectural characteristics. However, no-swizzling, pointer-at-a-time, and object-at-a-time swizzling schemes are independent of shipping granularity, because these swizzling schemes require only one object at a time.

The access method, shipping granularity, and OID representation are mainly concerned with transparency, performance, and flexibility, respectively. However, not one of these factors overwhelms the other two. That is, design policy is a matter of preference among these factors in terms of which one is chosen first. For example, one who considers transparency to be the most important may design and implement a memory protection based system. In the same way, one who attaches great importance to performance may choose a page

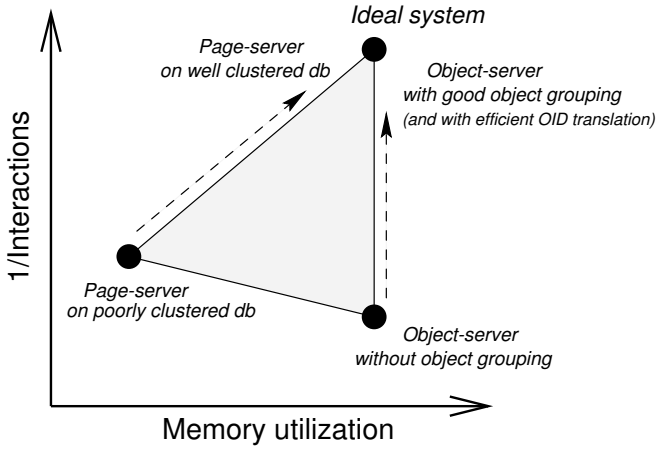


Fig. 9. A way to an ideal system: in performance

server architecture, and one who chooses flexibility will follow a logical OID scheme.

Although we cannot say that one of the three factors is the most important and it is not easy to evaluate these issues quantitatively, we can suggest criteria for an ideal system that is satisfactory from each of the above three angles.

First, concerning the performance factor, Figure 9 depicts the tradeoff between memory utilization and network interactions. The approximate locations of a page server and an object server are also indicated. An object server is good with respect to memory utilization at the expense of many client-server interactions. On the other hand, the location of a page server depends on clustering. In Figure 9, an ideal system is located where the memory utilization and 1/interaction are maximized. As already known, it is possible for page server based systems to approximate an ideal system through a good clustering technique. On the other hand, object server based systems try to achieve an ideal system through good logical grouping polices. An efficient mechanism of logical OID translation can also be an important means of improving the system.

Turning our attention to flexibility, physical OID makes object movements and replications very difficult [18]. Thus, systems that adopt a physical OID scheme can approach an ideal system through the effective mechanism of dynamic object movements and distributions.

Finally, from the point of transparency, more efficient and transparent access methods should be developed to drive non-memory-protection-based systems toward an ideal system. Also, operating system improvements can make a memory protection architecture more safe and attractive by providing efficient page trapping facilities.

7 Summary

In this paper, we have surveyed the current state-of-the-art of object storage systems by focusing on three main architectural issues: client-server architecture, object identifier representation, and object access method. In addition, we have briefly discussed transaction processing, object clustering, and object versioning that are related to the performance and the modeling power of object storage systems. The following summarizes the survey results.

In object-oriented databases, each object has its own unique identifier, which remains invariant, independent of the object's value and structure, throughout the lifetime of a database. To access a persistent object through OID, an object on the disk should be addressed and then brought into main memory first. It is the representation of OID that affects the performance of this process, since the cost of locating and fetching objects is mainly dependent on the implementation techniques of OID. Two kinds of OID representations are popular with object-oriented database systems: physical OID and logical OID.

In contrast to traditional database systems, OODBMSs usually ship data from a server to clients so that clients can navigate the shipped data and perform query processing locally by themselves. The important parameter here is the shipping granularity, that is, the number of objects that should be transferred together. The granularity of data shipping actually implies the overall system architecture of an object storage system. The two most commonly used data-shipping techniques are object server and page server.

Finally, the performance and transparency of access to objects, cached in a client, are determined by access method, such as the object fault handling mechanism and the pointer swizzling policy. Pointer swizzling can improve the performance of object access by skipping any lookup-table search, particularly in CPU-intensive applications. In particular, memory protection systems provide transparent and efficient access to persistent objects in the same manner as to transient objects.

The three main factors above – shipping granularity, OID representation, and access method – are mainly concerned with performance, flexibility, and transparency, respectively. Concerning the performance factor, object grouping is the most challenging area to improve the performance of object storage systems. A reasonably intelligent logical grouping will be attractive for systems based on object server architecture, while better techniques of physical grouping, that is, clustering, should be devised to enhance the performance of page server based systems. Also, more research into the implementation techniques of an object identifier and an efficient and transparent access method are needed to approximate more closely an ideal system that provides trans-

parency, high performance, and flexibility.

Table 3 shows overall survey results of this paper.

In the future, we would like to develop prototypes for alternative ways of implementing object storage systems by focusing on three main architectural issues, and to evaluate the corresponding performance of each alternative. We believe that such a study could make our current work more concrete and realistic. We are also interested in finding an analytic model that can predict the performance of object storage systems, according to the major architectural issues discussed in this paper.

Acknowledgments The authors would like to thank all those at OOPSLA Laboratory in the Department of Computer Engineering, Seoul National University who have been involved in the SOP project. We also wish to thank the anonymous referees for their valuable comments.

References

[1] R. Agrawal, S. J. Buroff, N. Gehani, and D. Shasha. Object Versioning in Ode. In *Proceedings of the International Conference on Data Engineering*, 1991.

[2] R. Agrawal, S. Dar, and N. Gehani. The O++ Database Programming Language: Implementation and Experience. In *Proceedings of the International Conference on Data Engineering*, 1993.

[3] R. Ahmed. Version Management of Composite Objects in CAD Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1991.

[4] J.-H. Ahn and H.-J. Kim. SEOF: An Adaptable Object Prefetch Policy For Object-Oriented Database Systems. In *Proceedings of the International Conference on Data Engineering*, 1997.

[5] R. Arlein, J. Gava, N. Gehani, and D. Lieuwen. *Ode 4.2(Ode<EOS>) User Manual.* AT&T Bell Laboratories, 1996.

[13] '–' indicates an item unknown at the time of writing of this paper.

[14] Ontos and Versant also support APIs to group-read objects or to obtain a closure of objects.

[15] All references to the newly activated objects from other objects already read are swizzled.

[16] A swizzled pointer points to an object descriptor (or fault block) instead of the object.

[17] Clustering objects of a single class is done automatically at class definition time, but a user may instruct the database to cluster the instances of a group of classes.

Table 3
Overall survey results [13]

Features	EXODUS	Itasca	O_2	Objectivity/DB	ObjectStore	Ode	Ontos	Versant
Architecture (Shipping)	Page	Object	Page	Page	Page	Page	Object [14]	Object [14]
OID representation	Physical (12 bytes)	Typed logical (12 bytes)	Physical (8 bytes)	Physical (8 bytes)	Physical (8 bytes)	Physical (8 bytes)	Pure logical (8 bytes)	Pure logical (8 bytes)
Object fetch (?-at-a-time)	Object	Object	Page	Page	Page	Page	Object	Object
Pointer swizzling (?-at-a-time)	Pointer	Pointer	Pointer	Pointer	Page	No swizzling	Pointer [15]	Object
Swizzling check	Edge marking [16]	Edge marking [16]	Edge marking	Edge marking	Memory protection		Edge marking	Fault block
Access interface	Call	Call	Call(O2C), Indirect(C++)	Indirect	Direct	Indirect	Call, Indirect	Call, Indirect(C++)
Nested transaction	No	Yes	No	No	Yes	No	Yes	Yes
Check-in/check-out	No	Yes	No	Yes	Yes	No	–	Yes
Cache coherence	Validation	Validation	Callback	Validation	Callback	2PL	–	Validation
Clustering (Granularity)	Object	Class [17], Set of classes	Set of classes	Object	Object	Class	–	Set of classes
Versioning (VDH type)	Tree	Tree	DAG	DAG	DAG	Tree	–	DAG

[6] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O2.* Morgan Kaufmann Publishers, Inc., 1992.

[7] V. Benzaken and C. Delobel. Enhancing Performance in a Persistent Object Store: Clustering Strategies in O2. In *Proceedings of the 4th International Workshop on Persistent Object Systems*, 1990.

[8] E. Bertino and L. Martino. *Object-Oriented Database Systems: Concepts and Architecture.* Addison-Wesley Publishing Company Inc., 1993.

[9] A. Biliris and E. Panagos. *EOS User's Manual Release 2.0.1.* AT&T Bell Laboratories, 1993.

[10] M. J. Carey, M. J. Franklin, M. Lyvny, and E. J. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1991.

[11] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[12] R. G. G. Cattel and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.

[13] R. G. G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley Publishing Company Inc., 1991.

[14] E. E. Chang and R. H. Katz. Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1989.

[15] J. R. Cheng and A. R. Hurson. Effective Clustering of Complex Objects In Object-Oriented Databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1991.

[16] J. R. Cheng and A. R. Hurson. On The Performance Issues of Object-Based Buffering. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1991.

[17] D. J. Dewitt and D. Maier. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Proceedings of the International Conference on Very Large Data Bases*, 1990.

[18] A. Eickler, C. A. Gerlhof, and D. Kossmann. A Performance Evaluation of OID Mapping Techniques. In *Proceedings of the International Conference on Very Large Data Bases*, 1995.

[19] EXODUS Project Group. EXODUS Storage Manager V3.0 Architectural Overview, 1993.

[20] EXODUS Project Group. Using the EXODUS Storage Manager V3.1, 1993.

[21] M. J. Franklin, M. J. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Trans. Database Syst.*, 22, 1997.

[22] C. A. Gerlhof, A. Kemper, and C. Moerkotte. On the Cost of Monitor-

ing and Reorganization of Object Bases for Clustering. *ACM SIGMOD Record*, 25(3), 1996.

[23] J. Gray and A. Reuter. *Transaction Processing.* Morgan Kaufmann Publishers, Inc., 1993.

[24] A. L. Hosking and J. E. B. Moss. Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications(OOPSLA)*, 1993.

[25] A. R. Hurson, S. H. Pakzad, and J. Cheng. Object-Oriented Database Management Systems: Evolution and Performance Issues. *IEEE Computer*, 1993.

[26] IBEX Object Systems, Inc. ITASCA Distributed Object Database Management System Technical Summary Release 2.3, 1995.

[27] IBEX Object Systems, Inc. ITASCA Technical Summary Release 2.3, 1995.

[28] R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 1990.

[29] A. Kemper and D. Kossmann. Dual-Buffering Strategies in Object Bases. In *Proceedings of the International Conference on Very Large Data Bases*, 1994.

[30] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB journal*, 4(3), 1995.

[31] A. Kemper and G. Moerkotte. *Object Oriented Database Management: Applications In Engineering and Computer Science.* Prentice Hall, 1994.

[32] S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications(OOPSLA)*, 1986.

[33] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[34] W. Kim. *Modern Database Systems – The Object Model, Interoperability, and Beyond.* Addison-Wesley Publishing Company Inc., 1995.

[35] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Trans. on Knowledge and Database Eng.*, 2(1), 1990.

[36] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. A Transaction Mechanism for Engineering Design Databases. In *Proceedings of the International Conference on Very Large Data Bases*, 1984.

[37] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Commun. ACM*, 34(10), 1991.

[38] M. E. Loomis. *OBJECT DATABASES - The Essentials.* Addison-Wesley Publishing Company Inc., 1994.

[39] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In B. Shrive and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* The MIT Press, 1987.

[40] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[41] J. E. B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Trans. Softw. Eng.*, 18(8), 1992.

[42] Object Design, Inc. ObjectStore Release 4.0 Online Documents, 1995.

[43] Object Design, Inc. Objectstore technical overview, 1997. http://www.odi.com/products/os/techovrwv.html.

[44] Objectivity, Inc. Objectivity/DB Technical Overview, Version 3, 1995.

[45] Objectivity, Inc. Objectivity technical overview, version 4, 1997. http://www.objectivity.com/Products/TechOv.html.

[46] Ontos, Inc. ONTOS Product Description, 1996.

[47] M. Palmer and S. B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the International Conference on Very Large Data Bases*, 1991.

[48] E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi. Client-Based Logging for High Performance Distributed Architecture. In *Proceedings of the International Conference on Data Engineering*, 1996.

[49] R. Ramarkishnan and D. J. Ram. Modeling Design Versions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 556–566, 1996.

[50] D. Schuh, M. J. Carey, and D. J. DeWitt. Persistence in E Revisited—Implementation Experiences. Technical Report #957, University of Wisconsin-Madison, 1990.

[51] E. Sciore. Versioning and Configuration Management in an Object-Oriented Data Model. *VLDB journal*, 3(1), 1994.

[52] E. Shekita and M. Zwilling. Cricket: A Mapped, Persistent Object Store. Technical Report Computer Sciences #956, University of Wisconsin-Madison, 1990.

[53] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, 1992.

[54] O_2 Technology. A Technical Overview of The O_2 System, 1994.

[55] O_2 Technology. O_2Engine Technical Features, 1997. http://www.o2tech.fr.

[56] M. M. Tsangaris and J. F. Naughton. On the Performance of Object Clustering Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1992.

[57] F. Vaughan and A. Dearle. Supporting Large Persistent Stores Using Conventional Hardware. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, San Minato, Italy, 1992.

[58] Versant Object Technology Corp. Versant ODBMS Release 4, 1996.

[59] Versant Object Technology Corp. Versant ODBMS Release 5, 1997. http://www.versant.com/products/rel5/index.html.

[60] Y. Wang and L. A. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of the ACM*

SIGMOD International Conference on Management of Data, 1991.

[61] S. J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin-Madison, 1994.

[62] S. J. White and D. J. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *Proceedings of the International Conference on Very Large Data Bases*, 1992.

[63] S. J. White and D. J. DeWitt. QuickStore: A High Performance Mapped Object Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[64] P. R. Wilson and S. V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, 1992.