A Performance Evaluation of OID Mapping Techniques

André Eickler* Carsten A. Gerlhof* Donald Kossmann†

*Universität Passau
Fakultät für Mathematik und Informatik
Lehrstuhl für Dialogorientierte Systeme
D-94030 Passau, Germany
[*eickler* | *gerlhof*]@db.fmi.uni-passau.de

†University of Maryland
Department of Computer Science
College Park, MD 20742 USA
kossmann@cs.umd.edu

Abstract

In this paper, three techniques to implement logical OIDs are thoroughly evaluated: hashing, B-trees and a technique called direct mapping. Among these three techniques, direct mapping is the most robust; it induces at most one page fault to map an OID, and it scales very well to large, rapidly growing databases. Furthermore, the clustering of handles that are used to map logical OIDs is studied. In particular, the performance of B-trees and direct mapping can improve significantly if the handles of objects that are frequently accessed by the same methods are clustered. For direct mapping, two placement policies are compared: linear and matrix clustering.

1 Introduction

The full support of object identity is one of the most important features of object-oriented database systems [KM94]. To improve referential integrity, an object base system allocates an object identifier (OID) to every object at the time the object is created. The OID is used to identify the object uniquely and to implement inter-object references. It is independent of the state of the object, and it is not changed during the whole life-time of the object [KC86, AK89]. In addition, most systems guarantee that even if an object is deleted, the OID generated for the object will never be used again to identify another object; i.e., OIDs of deleted objects are not reclaimed. To this end, OIDs are usually at least 8 bytes long, and the database system must support a 64-bit or even larger address space.

**Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995**

Basically, there are two kinds of OIDs: physical and logical OIDs. Both can be found in existing systems. A physical OID is constructed in such a way that it contains the permanent address of the object it refers to (i.e., the id of the disk, the page number and the slot). An object can directly be loaded from disk on the basis of a physical OID. On the negative side, the reorganization and reclustering of the database is difficult because an object cannot simply be moved to another page or another disk if physical OIDs are used. To move an object, a placeholder (i.e., a *forward*) must be established; but, these placeholders annihilate the advantage of physical OIDs as often two page faults are required to read an object on the basis of a physical OID and the placeholder, and the storage utilization is reduced as the placeholders of moved objects fragment the data pages.

This work makes a strong point for logical OIDs (also called surrogates). Logical OIDs are more flexible than physical OIDs, since they do not contain the permanent address of the objects they reference. Objects can, therefore, be moved freely, and thus, the database can well be reorganized if logical OIDs are used [GKKM93]. In addition, logical OIDs support object replication in a distributed system, object fragmentation, access to different versions of objects, and dynamic schema evolution more effectively than physical OIDs. The flexibility, however, must be paid for: before an object can be loaded from disk, its logical OID must be *mapped* to determine the permanent address of the object; i.e., the *handle* that contains the address of the object must be obtained. The mapping of logical OIDs can require additional page faults and reduce the performance of a system significantly if it is not implemented carefully.

To investigate the *price* of logical OIDs, three alternative mapping techniques are thoroughly evaluated in this paper: hashing, B-trees and a technique referred to as direct mapping. All three techniques fully support logical OIDs as defined in [KC86]; i.e., OIDs that are distinct and independent of the state, location, structure and behavior of the referenced objects. The performance experiments indicated that direct mapping outperforms both hashing and B-trees in every case. Direct mapping guarantees that at most one page fault is required to carry out the mapping, and it scales very well, even if the database grows rapidly. In addition, concurrency control and recovery is easy to implement, and no transactions are blocked if new mappings must be recorded when new objects are created, or mappings must be modified when objects are moved. Whereas hash tables and B-trees are general-purpose index structures, direct mapping was specifically designed to implement logical OIDs. Direct mapping exploits that OIDs are generated by the system (rather than being user defined) and encodes the location of the handle of an object into the logical OID of an object.

The performance experiments, furthermore, showed that the overhead to map logical OIDs is often very low. In particular, if the handles of logically related objects are clustered, logical OIDs have comparable retrieval performance to physical OIDs; i.e., the overhead to map logical OIDs is only marginal. To improve the clustering of handles using direct mapping, two placement policies for handles were devised and compared in detail: linear and matrix placement.

The remainder of this paper is organized as follows: Section 2 outlines alternative configurations of name servers. The name servers are the processes that generate OIDs, maintain the bindings of OIDs to objects and carry out the mapping of OIDs. Section 3 describes the three mapping techniques. In addition, this section discusses related work; i.e., how logical OIDs were implemented in a couple of sample systems. Section 4 proposes OID generation algorithms for the three mapping techniques and shows how handles are placed into handle pages. Section 5 defines the benchmark environment, and Section 6 summarizes the results of the performance experiments. Section 7 concludes this paper.

2 Name Server Configurations

As stated above, a name server generates OIDs and maintains the mappings between OIDs and objects. This section discusses how a name server can interact with clients and data servers, and it outlines how several name servers can coexist in a distributed system. The remainder of this paper, then, is focused to study alternative mapping techniques.

2.1 Separation of the Name Server

While there are only few approaches known so far to enhance name management in database systems [KW94], separated name servers are widely accepted in the area of operating systems [Wat81]. Here, many different forms of named objects coexist (such as files, environment variables and hosts) and it is useful to handle them in a uniform way.

Many database systems, however, integrate the name service in the data server. Though naming in databases is more uniform than in operating systems, it is, nevertheless, reasonable to separate the name server from the data server. Naming of objects is a very important task and needs to be as flexible as possible. In particular it should be tailored to the specific application. Open OODB [WB94], for example, provides facilities to select a suitable implementation for naming on a per-object basis. It could also be profitable to assign resources, such as disk drives or processors, specifically to the name service.

The configuration of a name server has two dimensions. The first dimension is the mapping technique used, which we are mainly concerned with in this paper. The second dimension is the coupling of clients, data servers and name servers. Possible solutions are depicted in Figure 1. (For simplicity, Figure 1 depicts only the situation where one name server is used in a client-server architecture.)

Figure 1a) shows a classical operating-system architecture: the clients directly communicate with both the name and the data server. A client gets a handle from the name server and passes it to the data server to retrieve the required object. A handle can be thought of as a "pointer" to the data. An example of this procedure can be found in the implementation of a file system. A name service is attached to every directory and is used to obtain the handle of every file located in the directory. The file server, then, provides the functionality to read a file on the basis of its handle.

Recent architectures, as found in the distributed operating system Spring [MGH$^+$94], do not give any handles or low-level identifiers to the client. To avoid potential security leaks, they directly return (a copy of) the requested object, as depicted in Figure 1b). Additionally, this architecture avoids race conditions that might occur in architecture 1a), for example, when one client obtains the handle of an object while another tries to move it. There is no connection between clients and data servers, and, therefore, a client is not aware which data server provides an object.

In database systems, however, clients must always be able to communicate with the data server. It is not reasonable to pass all information needed (e.g., for transaction management) through the name server. Therefore, the architecture sketched in Figure 1c) was used in this work. The name and the data server can run on the same workstation as indicated in Figure 1c) to avoid network communication; but, they can just as well run on different workstations.

2.2 Distributed Name Services

In general, several name servers can coexist to make objects accessible in a distributed environment. Many name services (like the ones used in DNS [Moc87], NIS+ [Sun93], Conch [KW94] and SHORE [CDF$^+$94]) use a hierarchically structured name space. Spring uses a general *naming graph* in which name servers can be arbitrarily connected.

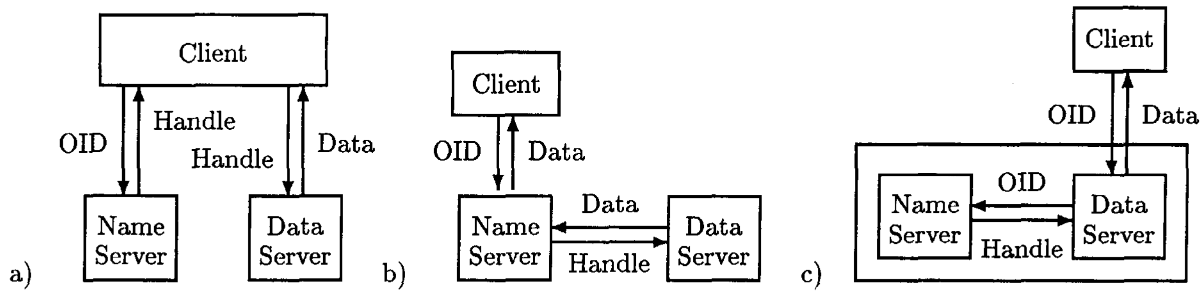


Figure 1: Configurations of Name Servers

Like the coupling of name servers to clients and data servers described in the previous section, the distribution architecture is independent of the mapping technique used by an individual name server. The mapping techniques described in the remainder of this paper can be used for any coupling and distribution architecture.

3 Mapping Techniques for Logical OIDs

In this section, various mapping techniques for logical OIDs are reviewed and discussed. The mapping function $f : OID \mapsto Handle$ determines for a given logical object identifier the physical address (i.e., the handle) of an object. In the following, two *indexed-based* mapping techniques and the *direct mapping* technique are discussed.

3.1 Index-Based Mapping

3.1.1 Hashing

At first glance, hashing seems to be the most attractive mapping technique since it has a constant access time independent of the number of entries in the hash table. Much work has been done in developing good hashing techniques, see [ED88] for an overview. Directory-based hashing techniques like [FNPS79] need two disk accesses in the worst case: one for the directory and one for the bucket. Directory-less hashing techniques like [Lar88b, Ahn93] try to retrieve entries with at most one disk access by compressing the split history in such a way that it can be kept in main memory. Other techniques like those described in [Lit80, Lar88a] avoid directories by temporary chaining overflow buckets which can be split at a later time.

An important tuning factor is the *hash function.* A good hash function ensures a uniform distribution which is crucial for every hashing technique devised so far. To map OIDs, a good hash function can easily be defined because all OIDs are generated by the name server. The performance of a hash function, however, can decrease if many objects are deleted.

In the following, we discuss the scalability, storage utilization, concurrency control, recovery, and replication of hashing-based mapping techniques. To make the comparison more comprehensive, all the mapping techniques will be discussed considering theses criterions. An important aspect which is not explained here, but crucially affects the performance is the clustering of handles into pages. This topic is discussed in detail in Section 4.

• **Scalability.** The most serious problem with hash tables is that they do not scale very well. If the size of a hash table can be estimated in advance, the hash table can be tuned so that overflows of buckets are rare. In database systems, however, the size of the database often grows in a short period of time. In this case, a hash table degrades fast with the number of bucket splits resulting in expensive reorganizations. Using techniques which compress the split history, CPU processing becomes a dominant factor when many splits occur. In conclusion, the initial size of a hash table is an important (and rather difficult) tuning factor.

• **Storage Utilization.** The analysis of several dynamic hashing schemes has indicated that in an average case, 69% of the capacity of the buckets is used [ED88]. If objects are rarely deleted and the hash table is tuned given the precise size of the object base, the storage utilization of an OID-mapping hash table is usually much higher because the dedicated hash function and the key generation algorithm fill the buckets uniformly.

• **Concurrency Control and Recovery.** Only little has been reported on concurrency control and recovery techniques for hash tables and how they affect the performance of the system [GR93]. It is not clear if simple value logging is sufficient, or if sophisticated but more complicated techniques like [Moh92] are needed for a mapping technique based on hashing.

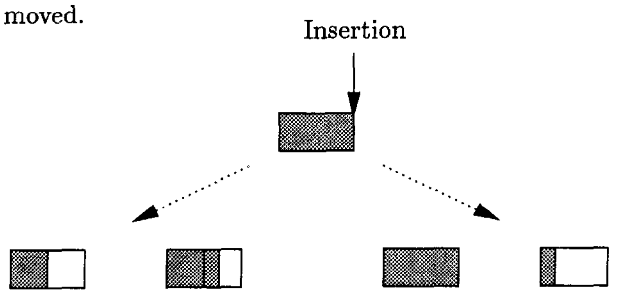
• **Replication.** In an architecture with more than one name server (e.g., a peer-to-peer architecture), handles can be replicated on several name servers. A logical OID and the corresponding handle can be recorded by every name server using hashing regardless of where the logical OID was generated. When an object is deleted or moved to another location, all the name servers that keep a replica of the handle of the object are informed.

Versant and Itasca are examples of commercial systems that use hash tables to map OIDs. Itasca [Ita93] provides a hash table for every object class separately. Itasca's way of mapping OIDs is, thus, carried out in two steps: first, the class name—which is encoded in the OID—is hashed to get the appropriate hash table, and then, the per-class hash table is consulted to get the object's handle. Since the number of classes is usually relatively small, the whole class-name hash table can be kept in main memory.

3.1.2 B-Tree

The initial size of a hash table is an important tuning parameter. B-trees [BM72, Com79] have no such tuning parameter and are, therefore, much simpler to use in practice. At first glance, it appears that they cannot compete with the retrieval performance of hashing, since the whole path from the root down to the leaf nodes must be read to map an OID. In a B^+-tree of height three, however, several million OIDs and their corresponding handles can be stored, and a B^+-tree of height four contains the mapping information of almost one billion objects. If the first two levels can be cached in main memory by the name server, at most two disk accesses are required to retrieve a handle [COL92].

For the purpose of mapping OIDs, a specially tuned implementation of B-trees was used. The performance of a general-purpose B-tree is at its worst if keys are inserted in an ascending order: splits occur often and 50% of the storage space are wasted (Figure 2a). Logical OIDs, however, are often allocated in an ascending order; for example the first OID generated could have the number "0," the second the number "1" and so forth. Therefore, the method shown in Figure 2b) was used. Rather than moving half of the entries of an over-full node into the new node, only the last entry is moved.



a) Split of a General-Purpose B-Tree　　b) Tuned Split for OID Mapping

Figure 2: Splitting of a Leaf Page in a B-Tree

In the following, we discuss some aspects of the B-tree-based mapping technique in more detail:

- **Scalability.** B-trees have logarithmic properties. Due to the uniform way in which mappings are inserted, B-trees scale much better than most hashing schemes in large, growing databases.
- **Storage Utilization.** Using the tuned split method, a B-tree for mapping OIDs can have almost 100% storage utilization if no objects are deleted. In addition, prefix compression can be exploited so that more mappings can be recorded in the leaf of a B^+-tree than in the bucket of a hash table. Obviously storage utilization decreases if many objects are deleted from the object base and the corresponding mappings must be destroyed. In rare cases, the prefix compression of OIDs can then be counterproductive.
- **Concurrency Control and Recovery.** Index structures like B-trees have proved to be a serious bottleneck of the system if they are updated. Several techniques to improve concurrency and recovery have been proposed [ML92, SO92]. Nevertheless, the implementation of these algorithms is difficult and frequent modifications can reduce the performance of the system significantly in any case.
- **Replication.** The replication of mappings can easily be carried out — in the same way as using hash tables.

B-trees are used in GemStone [MS87] and in SHORE [CDF+94] to implement logical OIDs.

3.2 Direct Mapping

Index-based mapping techniques usually find handles by *comparing* OIDs. In this section, we discuss a technique that encodes the address of the handle into the logical OID. The rational behind this approach is that handles can be directly accessed without the help of an additional index structure—therefore, this technique is referred to as direct mapping.

To support direct mapping, the handles are organized in extensible arrays, so-called *handle segments.* A handle segment is a list of handle pages, and every handle page contains slots with a handle and a *unique* field each. Using direct mapping, an OID has the following layout: creation site (two bytes), handle page number (four bytes), slot number (two bytes), and a *unique* field (four bytes). The creation site, page and slot numbers are used to find the handle of an object. The *unique* field is used to detect dangling references; for a valid OID it must match the *unique* field of the corresponding slot in the handle page. If an object is removed from the database, the name server increments the *unique* field of the slot in the handle segment and, thus, implicitly invalidates all the dangling references to that object. The name server can, then, safely re-use the slot to record the mapping of a new object.

Figure 3 shows a handle segment with a page capacity of five handles and an object segment with pages containing two objects. Handles are represented by pointers to emphasize the fact that handles are physical addresses. Empty slots of a handle segment are recorded in a *free space bitmap.* Upon deletion of an object, the slot of its handle is marked as free in the bitmap, and upon creation of a new object, a free handle slot is allocated from the bitmap.[1]

Note that direct mapping fully supports logical OIDs: for example, a data segment can be reorganized,

[1] In our prototype, the bitmap keeps a bit per slot; the bitmap could, however, also record handle pages with free slots.

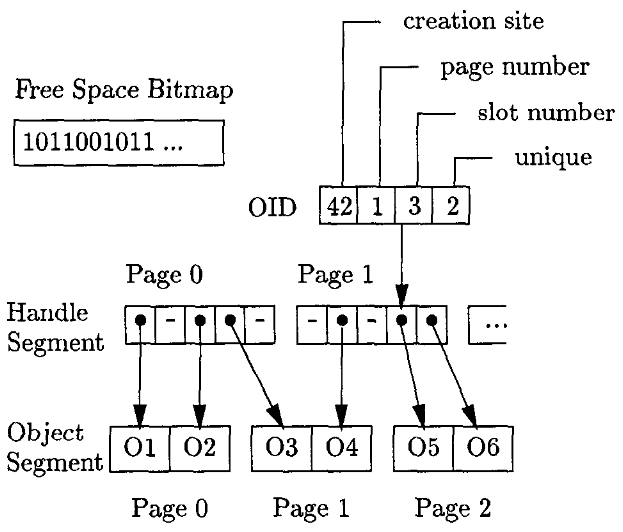


Figure 3: The Handle Segment in the Direct Mapping Technique

and objects can be moved by updating the handles of objects only. In comparison to the index-based techniques, the main characteristics of direct mapping are listed in the following:

• **Scalability**. Handle segments can easily be extended without any reorganization costs. Therefore, allocation costs are independent of the number of objects stored in the database. In addition, an OID can be mapped with at most one page fault independent of the size of the database.

• **Storage Utilization**. Handle pages can be filled so that almost 100% storage utilization can be achieved. If an object is deleted, the corresponding empty slot in the handle segment can be re-used using the free space bitmap. Using direct mapping, only the handle and the unique identifier of the object (rather than the whole OID) must be recorded in a handle segment. Therefore, more OIDs can be mapped using a single handle page than with a B-tree or a hash table.

• **Concurrency Control and Recovery**. No directory structures are needed to map OIDs. Thus, no special concurrency control and recovery is required. Any fine-grained recovery technique like ARIES [MHL+92] can be used, and the creation of new objects does not block active transactions that only read and modify existing objects.

• **Replication**. Handle pages can be replicated in a distributed system in the same way as ordinary data pages. Whereas replication is fine-grained using hashing or B-trees (i.e., the handles of individual objects are replicated), it appears that handle pages are the most favorable unit of replication for direct mapping.

Techniques similar to direct mapping have been used in a couple of research prototypes [HZ87, BR90, WW90]; but, we know of no commercial system that exploits direct mapping.

4 Clustering of Handles

In this section, the clustering of handles is discussed. That is, how the handles of objects are placed into disk pages at the time of creation so that page faults for mapping logical OIDs are reduced in a name server. For example, handles of objects whose logical OIDs must frequently be mapped should be placed into the same pages. In addition, it is discussed why the reorganization and reclustering of handle segments must be precluded regardless of which mapping technique is used so that the initial placement of the handles is the only opportunity to achieve good clustering performance.

4.1 Placing Handles into Handle Pages

4.1.1 Index-Based Mapping

Exploiting clustering is very difficult using hash tables because the hash function determines in which buckets to place handles. To place the handles of logically related objects into the same bucket, an order-preserving hash function, e.g., tries [Lit88, Oto88, Chu92], and a special key generation algorithm is required. Such an approach was used in the design of the database system GRAS [KSW92]. Given the OID of a related object, the key generation algorithm tries to generate a new OID whose mapping is recorded in the same bucket. At the same time, the mappings of related objects should still be stored in the same bucket after a split has occured. Unfortunately, such an approach is not viable in large databases with many objects: the hash tables tend to degenerate because some buckets must frequently be split whereas other buckets are not used. Therefore, only hash functions that spread the handles among the available buckets as uniformly as possible were used in this study. OIDs were generated in an incremental manner (e.g., if object 1065 was just created, the logical OID of the next object that will be created will be 1066). In this approach, clustering cannot be exploited; but, the hash tables can easily be tuned if the size of the database is known.

As opposed to hashing, clustering can well be exploited in a B-tree using a sophisticated key generation algorithm. Given a "near" hint, i.e., the OID of a related object, the leaf page that records the mapping of the related object is looked up, and the OID of the new object is generated so that the handle of the new object is recorded in the leaf page. If the favored leaf page is full, the new mapping is recorded in a new (logically) adjacent leaf page according to the tuned split policy described in Section 3.1.2.

4.1.2 Direct Mapping

Using direct mapping, a name server has the flexibility to place the handle of a new object on a new, empty handle page or on any existing handle page that still has a free slot. In particular, just like with B-trees, the handles can be clustered into handle pages taking

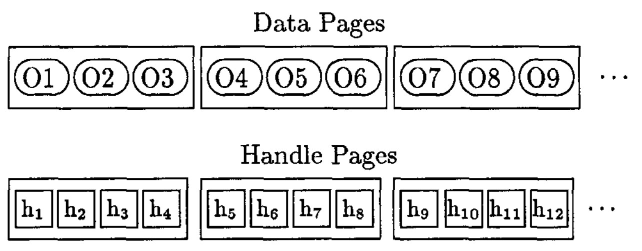


Figure 4: Linear Placement of Handles

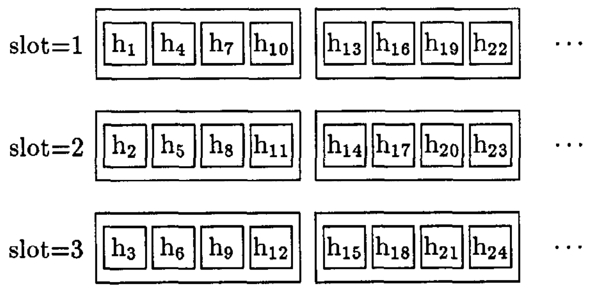


Figure 5: Matrix Placement of Handles (Handles refer to the data pages shown in Figure 4.)

into account how the corresponding objects are placed into data pages. The handles of logically related objects that were placed into the same partition of the database can, therefore, be clustered together, and a large set of objects can be read inducing only a few additional page faults to map their logical OIDs.

Two different approaches to place handles into handle pages were investigated in this work: handles were either clustered linearly or following a matrix scheme. Ideal linear clustering of handles is shown in Figure 4. Handles are placed into handle pages in the same order as the corresponding objects are placed into the data pages. As a consequence, the handles of objects that are located in the same data page are located in the same or an adjacent handle page.

If a page-server is used to ship data to the clients, the linear placement of handles can often be improved because subsequent requests to handles referring to objects that are located in the same page are very seldom. When an application requests the handle of an object from the name server to read the object, it will get the handles of all the other objects located in the page by the page-server "for free" without asking the name server [CDF+94].

To avoid handles of objects located in the same data page of being placed into the same handle page, the matrix scheme depicted in Figure 5 was devised. The handle pages are organized in logical files. The handle of the *first* object of every data page (i.e., the object stored in the first slot) is placed into a handle page of the first file; the handle of the *second* object of a page is placed into a page of the second file; and so on.

Matrix placement significantly outperforms linear placement if objects are read sequentially. For example, if the objects *O1* to *O9* are read sequentially, and only the OIDs *O1, O4* and *O7* must be mapped, the name server accesses only one handle page if matrix placement is used (cf. Figure 5) as compared to two handle pages with linear placement (cf. Figure 4). On the other hand, linear placement outperforms matrix placement if the OIDs of objects located in a few data pages and in different slots must be mapped. For example, if the OIDs *O6* and *O7* must be mapped, the name server accesses one handle page with linear placement and two handle pages with matrix placement.

Both the linear and the matrix placement of handles rely like the B-tree on the quality of the initial placement of objects into data pages. Of course, the handles will not be clustered very well if the corresponding objects are not clustered well. But, taking the placement of objects into account is definitely the best bet to cluster handles.

4.2 Reorganization

The handle pages cannot be reorganized to improve clustering regardless of whether hashing, B-trees or direct mapping is used in a name server. In particular, handle segments cannot be reorganized if the corresponding data segments are reorganized. If direct mapping is used, a handle must be stored in the page in which it was placed at time of creation until the corresponding object is deleted from the object base, since references to the object (logical OIDs) directly refer to the handle in that page. As a consequence, the clustering properties of a handle segment described in the previous section (e.g., linearity) are not preserved if the corresponding data segment is reorganized. If the data segments are fairly small, however, a handle page will, nevertheless, contain many handles of logically related objects after a reorganization of a data segment.

Using hashing, the hash function determines in which bucket (handle page) a handle is stored. It is, in general, impossible to find a (new) hash function that places handles that are requested by the same methods into the same buckets. If a B-tree is used to map logical OIDs, the handles of adjacent logical OIDs are stored in the same leaf page of the B-tree; the *re-ordering* of entries and, thus, the allocation of a new OID for an object, must be precluded like in the other mapping techniques.

5 Benchmark Environment

5.1 Software Used

The performance of an isolated, centralized name server was analyzed in a client/page-server system; i.e., there was only one name server running, and this name server was separated from the page-server. At page-fault time, the clients passed the logical OID of an object to the page-server. The page-server called the name server which was run on the same workstation to map the OID and determine the home page of the

	number of objects	size of the database	size of the mapping file LH	BT	DM
small	200,000	40 MB	13 MB	9 MB	5 MB
large	5,000,000	1 GB	252 MB	204 MB	118 MB

Table 1: Characteristics of the Small and the Large Database

	Sequential	Random	Hot
Pages per module (n)	5,000	50	500
Reference pattern	scan	select, $r = 5$	select, $r = 500$
Number of hot modules	0%	0%	5%
Probability of choosing a hot module	0%	0%	80%

Table 2: Workload Parameters

requested object. Then, the page-server shipped the home page of the requested object together with the handles of all the other objects located in the page to the client.

Three different mapping techniques were studied:

(1) Linear Hashing. For the performance experiments, linear dynamic hashing was used in the same way as described in [Lit80]. The hash table was tuned for every object base individually given the (a-priori) knowledge of how many OIDs had to be recorded. As a consequence, the storage utilization of the hash table was almost 100% in all the experiments; no overflow buckets were required; and an OID could be mapped with at most one page fault. A bucket of the hash table contained at most 80 entries.

(2) B-Tree. B^+-trees were used [BM72]. A leaf of the B^+-tree always held more than 80 entries (usually about 100), since prefix compression was enabled and the tuned split policy devised in Section 3.1.2 was used. The height was three for the large database and two for the small one (the small and the large database are described in the following section).

(3) Direct Mapping. Using direct mapping, a handle page contained at most 170 entries because only the handles and *unique* fields of objects had to be stored.

Regardless of which mapping technique was used, all the logical OIDs were 12 bytes long.

5.2 Benchmark Used

To verify the scalability of the mapping techniques, the experiments were carried out against a large object base with 5 million objects and a small object base with 200,000 objects. The size of a page was 4K and every page contained 20 objects. Thus, the large object base had 250,000 pages (1 GB) and the small one 10,000 (40 MB). Table 1 summarizes the characteristics of the small and the large database.

The databases were divided into m modules. Every module consisted of $n = p/m$ pages and $n \cdot 20$ objects with $p =$ 250,000 for the large database and $p =$ 10,000 for the small database, respectively. Two different reference patterns were taken into account:

- **scan.** An application read sequentially all the objects of a module. In this operation, the name server was called n times, once for the first object of every page; the handles of the other objects located in the page were provided by the page-server when the page was shipped to the application.

- **select r.** An application read r different random pages of the module and accessed an arbitrary object of every page. In all, the name server was called r times for every module.

Table 2 defines the three different workloads that were studied. In every workload, transactions were submitted serially to the system. Every transaction operated on one module and carried out either a *scan* or a *select* on the module—depending on the workload. The number of page faults per request caused by the name server was recorded depending on the number of main-memory buffer frames that were available to cache handles.

In the *Sequential* and *Random* workloads every module was used by a transaction with the same probability. The *Sequential* workload scanned large modules, while *Random* navigated in small modules. The third workload, *Hot*, measured the influence of locality in the access profile. 5% of the modules were "hot" and used by 80% of the transactions.

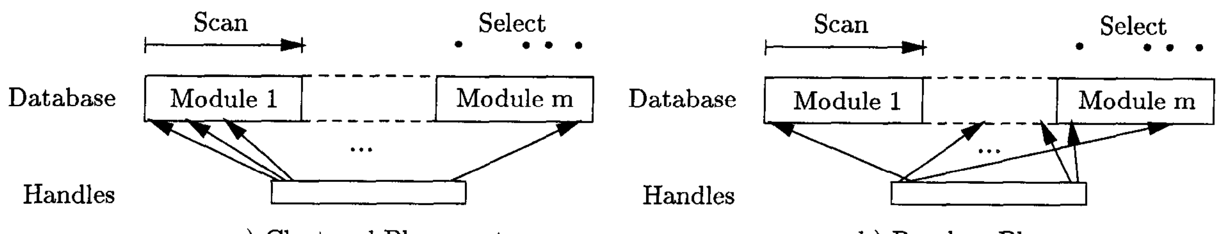
For each workload, the following placements of handles were considered:

- **Linear Hashing.** In the case of Linear Hashing (LH), the placement was dictated by the hash function, which distributed the handles uniformly over the buckets.

- **B-Tree.** The handles were placed according to the object placement (cBT) and randomly (rBT).

- **Direct Mapping.** Linear (clDM) and matrix clustering (cmDM) were used as described in Section 4, as well as a random placement (rDM).

The benchmark is illustrated in Figure 6. Figure 6a) shows the clustered placement of handles, which is the best case that can be achieved. Figure 6b) shows the random placement, where the handles have a worst



a) Clustered Placement b) Random Placement

Figure 6: Illustration of the Benchmark

case clustering. This worst case clustering occurs, for example, when the object base has been severely reorganized.

6 Performance Analysis

The performance analysis was divided into three parts. First, the retrieval performance of all the mapping techniques was compared using the three workloads described in the previous section. Second, the two placement policies for direct mapping were analyzed. Third, the recording of new mappings, i.e., the creation of new objects, was investigated.

6.1 Comparison of the Mapping Techniques

The results using the *Sequential* workload are presented in Figure 7. The left plot, Figure 7a), was obtained using a small database; the right one, Figure 7b), using a large database. In both figures, the number of buffer frames used by the name server is plotted against the X-axis. For the large database, the range was from 500 pages (2 MB) to 3000 pages (12 MB). In proportion to the size of the mapping files, the same range of buffer sizes were studied with the small and with the large database.

Linear Hashing had almost constant retrieval performance regardless of how much buffer was available. The accesses were evenly distributed over the hash table, which had about 63,000 pages in the large database. The effect of buffering on the performance was not significant because only a small fraction of the total table could be cached (e.g., 5% if 3,000 buffer frames were available). The B^+-tree with randomly clustered handles (rBT), on the other hand, took advantage of an increasing number of buffer frames by caching more internal nodes of the first levels. If the buffer was very small compared to the size of the B^+-tree, however, the B^+-tree was outperformed by the hash table. If the handles were clustered randomly, direct mapping performed a little better than both hashing and the B^+-tree because it used significantly less disk space (about 30,000 pages).

If the handles were clustered, the strength of B-trees and direct mapping was exhibited. Not all the inner nodes of the B^+-tree had to be cached to achieve good performance in this case, since only a single, *narrow* path from the root to the leaves had to be descended for each module. The leaves occupied about 1,000 pages per module, which resulted in 1,000 page faults/5,000 requests per module = 0.2 page faults per request. Analogously, direct mapping with linear placement (clDM) occupied about 600 pages per module and had therefore about 0.12 page faults per request (600/5,000 = 0.12). The best performance was achieved by using direct mapping with a matrix placement of handles. Here, all the relevant handles of a module that were required in the *scan* operation were clustered together into 30 handle pages.

The plots for the small and the large database are almost identical. No anomalies could be observed for the large database. This observation was also made for the *Random* and *Hot* workload.

In the *Random* workload, it was significantly more difficult to take advantage of the clustering of an object base. Therefore, the performance of direct mapping and B-trees with clustered handles decreased as compared to their performance in the *Sequential* workload. Nevertheless, direct mapping and B-trees profited from clustering, since the handles for a single module occupied only few adjacent pages. The results for the *Random* workload are shown in Figure 8a) (small database) and Figure 8b) (large database). Note that direct mapping with a matrix placement performed worse than with a linear placement of handles. This phenomenon is examined in detail in the next section.

The possibility to cache the inner nodes of a B-tree was more important in the *Random* than in the *Sequential* workload. This is documented by the slight bend in the curve for the clustered B-tree at the point where a significant portion of the inner nodes could be cached. The hash table had almost the same performance as in the *Sequential* workload.

The graphs obtained from the *Hot* workload (Figure 9a) and Figure 9b) have almost the same shape as the graphs obtained from the previous two workloads. This workload demonstrated an important property: long-term caching of handles that are used by many applications is often neglect-able. The performance of a mapping technique is completely dominated by the effect of clustering and the effort to *load* handle pages to serve a single application. If the handles are not clustered well, it is not possible to buffer the handles of a small subset of the database with a reasonable amount of buffer space. In the *Hot* workload, even the handles of the 5% *hot* objects could not be cached. Furthermore, direct mapping and the B-trees had al-

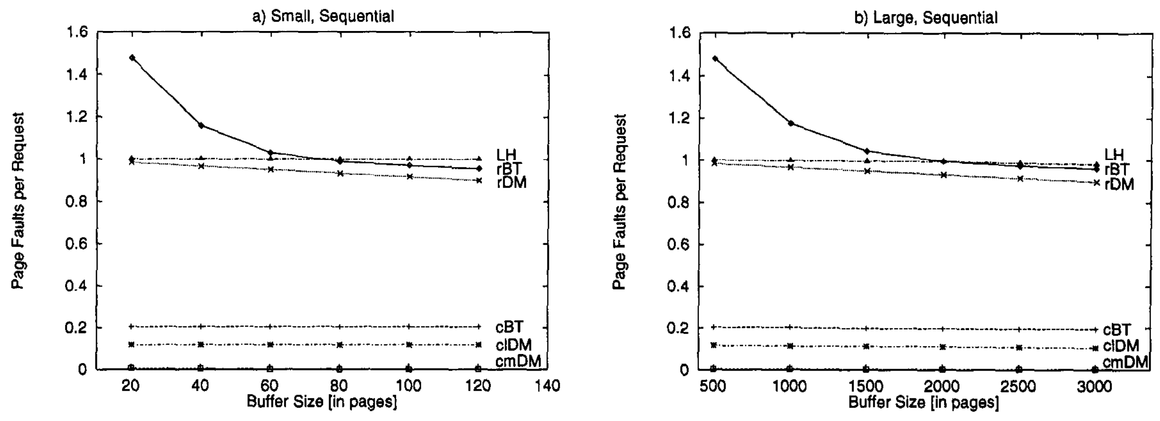


Figure 7: The *Sequential* Workload

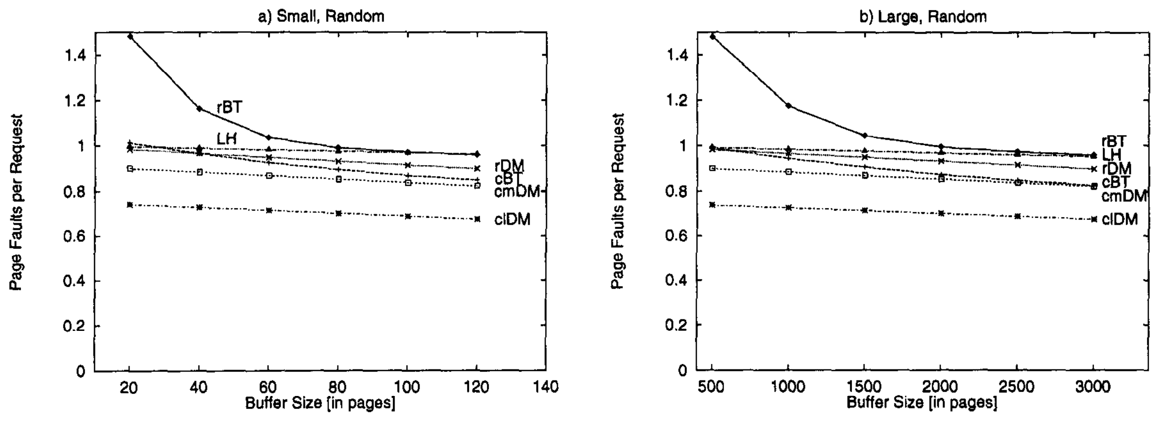


Figure 8: The *Random* Workload

most the same performance as in the *Sequential* workload, although the reference pattern was completely different.

The sharp bends for the clustered B-tree and direct mapping in Figure 9a) show the point at which the handles of the objects of one module could completely be buffered by the name server. When a module was loaded by an application, no additional page faults were required to re-access a handle page.

In summary, direct mapping outperformed the other two techniques in all cases. No clear winner could be identified between B$^+$-trees and linear hashing: if there was only little buffer space available and the handles were not clustered well, hashing performed better. Otherwise, the B$^+$-trees had less page faults per request. Here, we must keep in mind that the hash tables were perfectly tuned; the initial size was determined individually for the small and the large database.

6.2 Clustering Performance of Direct Mapping

To compare the linear and matrix placement policies, a second set of experiments was carried out using direct mapping only. The *Sequential* and *Random* workloads were run against the large database (i.e., 5 million objects) with a varying number of pages per module. 1,000 Pages main-memory buffer space were used by the name server to cache handles. Figure 10 summarizes the results.

If the handles were placed randomly into the handle pages (the rDM curves), a little less than one page fault per request was required independent of the module size and the reference pattern. If the handles of a module were clustered (the clDM and cmDM curves), the performance of direct mapping improved with the size of a module. This effect will be illustrated by the following example.

The 100 handles corresponding to a module with 5 pages and 100 objects were located in one or two handle pages with a capacity of 170 handles using the linear placement policy. Approximately, 1.5/5 page faults per request could, therefore, be observed when an application loaded a module (see the clDM curves). On the other hand, to load a module with 1,000 pages and 20,000 objects, 20,000/170 = 118 handle pages had to be read and only 118/1,000 page faults per request could be observed.

Working with large modules was more important if the handles of a module were clustered and the matrix placement policy was used in the *Random* workload. The handles of a module with 5 pages were placed into 20 handle pages to avoid that the handles of objects that were located in the same data page were located

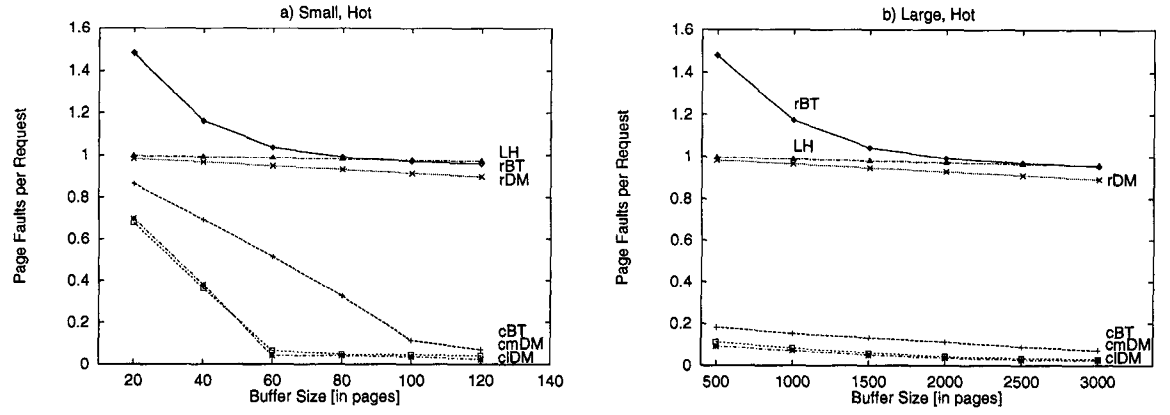


Figure 9: The *Hot* Workload

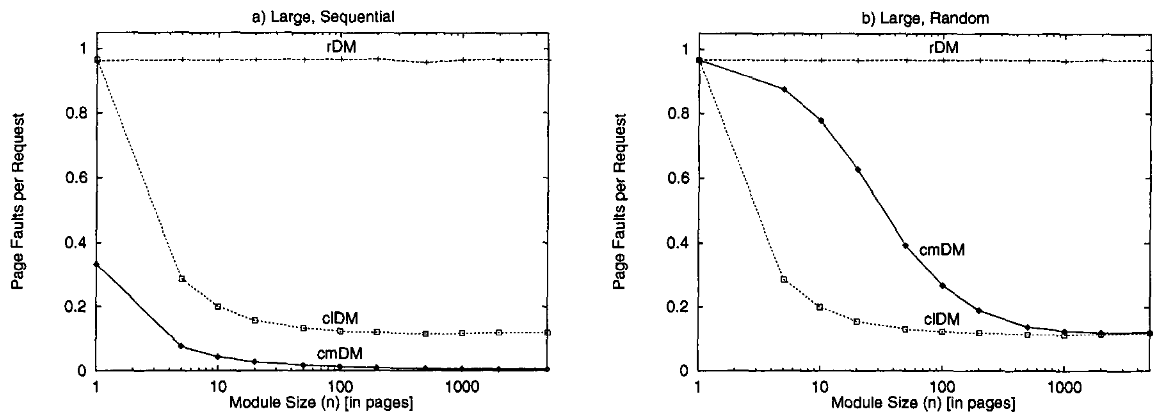


Figure 10: Linear vs. Matrix Placement

in the same handle page. Usually at least 4 of these 5 handle pages had to be accessed to carry out the *select* operation on a module, and more than 4/5 page faults per request could be observed. If the modules were large, however, the handles of a module could be placed into a small number of handle pages, and the property of matrix placement could be achieved at the same time.

Unlike the linear placement policy, matrix placement profited from the particular reference pattern of the *scan* operation. The performance of direct mapping with a linear placement of handles was independent of the reference pattern in which the objects of a module were accessed. The performance of the matrix placement policy, on the other hand, improved if the OIDs of objects located in some particular slots of the data pages must be mapped very often. In the *scan* operation, only the OIDs of the objects located in the first slot of a data page were mapped, and thus, direct mapping with linear placement showed up to 25 times as many page faults per request than direct mapping with matrix placement in the *Sequential* workload with large modules. Skew in the retrieval of objects from certain slots can also be observed if not all the objects have the same size and some quite large objects exist as well. In this case, objects will also more often be retrieved from the first slot of a page than from the tenth slot because many pages do not contain 10 objects.

6.3 Recording New Mappings

Table 3 studies the performance of a name server when new objects are created, and thus, new mappings must be recorded. New objects were placed into new data pages of the large database; again, a new data page contained exactly 20 objects before another data page was allocated to hold new objects. After the creation of an object, a large number of random retrieval operations was carried out using a buffer with 1000 pages.

Method	Faults/New Object
Direct Mapping	0.006
B-Tree	0.01
Linear Hashing	1
LH (after first split)	2.2

Table 3: Creating New Objects

For the B-tree, the OIDs were generated in an ascending order. The leaf page into which a new mapping was recorded last was kept main-memory resident. In addition, the buffer was large enough to keep the relevant inner nodes while the retrievals were carried out. Since the tuned split policy described in Section 3.1.2 was used and approximately 100 mappings could be recorded in a leaf page, one new leaf page per

100 new objects had to be allocated (considered as a page fault in Table 3), and no additional page faults to read nodes of the B-tree were required.

Analogously, the last 20 pages used to record new handles were kept main-memory resident using direct mapping with the matrix placement policy. Since 170 handles could be placed into a handle page, 1/170=0.006 page faults per new object could be observed.

For linear dynamic hashing, the hash function tried to place the mappings of the new objects uniformly into the existing buckets. As a consequence, approximately one page fault could be observed if no overflows occured, just as in the retrieval benchmarks. After the first overflow was generated and the hash table was split using Litwin's original criterion [Lit80], 2.2 page faults were required on an average to record a new mapping.

7 Conclusions and Future Work

In this work, three techniques to implement logical OIDs were investigated. Of these three techniques, direct mapping is the most robust. Direct mapping scales well in large object bases, and at most one page fault is required to map an OID; recording a new mapping is very cheap and seldomly induces a page fault. In addition, no applications are blocked by the name server if new objects are created or existing objects are moved. If separated and given enough resources (for example, a disk drive could be reserved to store the mapping information), a name server using direct mapping, therefore, is usually not the bottleneck of a system. For example, a disk drive could be reserved to store the mapping information.

Between hashing and B-trees, no clear winner can be identified. If much main-memory buffer space is available for the name server, a B-tree has the same or better performance than a hash table. If only little buffer is available, the B-tree is often outperformed by a tuned hash table.

The performance of B-trees and direct mapping improves dramatically when the handles of logically related objects are clustered. For direct mapping, two placement policies were devised and compared:

- **linear clustering:** the handles are placed into handle pages in the same order as the corresponding objects are placed into the data pages;
- **matrix clustering:** the handles of objects that are located in the same data page are never placed into the same handle page.

In a system that uses a page-server to ship data, much of the potential of clustering handles is wasted using the linear placement policy. Matrix clustering has at least the same performance as linear clustering if the locality sets accessed in the object base were fairly large. For particular reference patterns, it significantly outperforms linear clustering.

In future work, we intend to investigate the bulk loading of objects and the bulk allocation of new OIDs in detail. If OIDs are recorded and mapped in bulk—instead of one by one—additional tuning can be effected regardless of which mapping technique is used.

Acknowledgments. We would like to thank Alfons Kemper for his continuous support and his comments on a draft of this paper. Stefan Augustin implemented the B^+-tree with prefix compression, bulk-loading and other important features. Uwe Röhm implemented and tuned linear dynamic hashing. We would also like to thank Jeff Galarneau from Itasca Systems, Inc., for giving us details on the mapping technique used in Itasca. This work was supported by the German Research Council DFG under contract Ke 401/6-2 and by the Humboldt-Stiftung.

References

[Ahn93] I. Ahn. Filtered hashing. In *Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, volume 730 of *Lecture Notes in Computer Science (LNCS)*, pages 85–100, Chicago, IL, October 1993. Springer.

[AK89] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 159–173, Portland, OR, USA, May 1989.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[BR90] A. Brown and J. Rosenberg. Persistent object stores: An implementation technique. In Dearle et al. [DSZ90], pages 199–212.

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–394, Minneapolis, MI, USA, May 1994.

[Chu92] S. M. Chung. Indexed extendible hashing. *Information Processing Letters*, 44(1):1–6, 1992.

[COL92] C. Y. Chan, B. C. Ooi, and H. Lu. Extensible buffer management of indexes. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 444–454, Vancouver, Canada, 1992.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[DSZ90] A. Dearle, G. Shaw, and S. Zdonik, editors. *Implementing Persistent Object Bases, Principles and Practice, Proc. of the 4th International Workshop on Persistent Object Systems, Their Design, Implementation and Use*, Martha's Vineyard, September 1990. Morgan Kaufmann.

[ED88] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20(2):85–113, June 1988.

[FNPS79] R. Fagin, J. Nievergelt, J. Pippenger, and H. Strong. Extendible hashing—A fast access method for dynamic files. *ACM Trans. on Database Systems*, 4(3):315–344, 1979.

[GKKM93] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. In *Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, volume 730 of *Lecture Notes in Computer Science (LNCS)*, pages 301–316, Chicago, IL, October 1993. Springer.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann Publ. Co., San Mateo, CA, USA, 1993.

[HZ87] M. Hornick and S. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Trans. Office Inf. Syst.*, 5(1):70–95, January 1987.

[Ita93] Itasca Systems Inc. Technical summary for release 2.2, 1993. Itasca Systems, Inc., 7850 Metro Drive, Mineapolis, MN 55425, USA.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 408–416, November 1986.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[KSW92] N. Kiesel, A. Schürr, and B. Westfechtel. Design and evaluation of GRAS, a graph-oriented database system for engineering applications. Technical Report 92-44, RWTH Aachen, D-52056 Aachen, 1992.

[KW94] A. Kaplan and J. C. Wileden. Conch: Experimenting with enhanced name management for persistent object systems. In *Proc. of the Intl. Workshop on Persistent Object Systems (POS)*, Workshops in Computing, pages 318–331, Tarascon, France, September 1994. Springer.

[Lar88a] P.-Å. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.

[Lar88b] P.-Å. Larson. Linear hashing with separators—A dynamic hashing scheme achieving one-access retrieval. *ACM Trans. on Database Systems*, 13(3):366–388, September 1988.

[Lit80] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, 1980.

[Lit88] W. Litwin. Trie hashing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 19–29, Chicago, IL, USA, May 1988.

[MGH+94] J. Mitchell, J. Gibbons, G. Hamilton, P. Kessler, Y. Khalidi, P. Kougiouris, P. Madany, M. Nelson, M. Powell, and S. Radia. An overview of the Spring system, 1994.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems*, 17(1), March 1992.

[ML92] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 371–380, San Diego, USA, June 1992.

[Moc87] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, USC/Information Sciences Institute, November 1987.

[Moh92] C. Mohan. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. Technical Report RJ8682, IBM Almaden Research Center, March 1992.

[MS87] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392, Cambridge, MA, 1987. MIT Press.

[Oto88] E. J. Otoo. Linearizing the directory growth in order preserving extendible hashing. In *Proc. IEEE Conf. on Data Engineering*, pages 580–588, Los Angeles, CA, USA, 1988.

[SO92] M. Sullivan and M. Olson. An index implementation supporting fast recovery for the POSTGRES storage system. In *Proc. IEEE Conf. on Data Engineering*, pages 293–300, Tempe, AR, February 1992.

[Sun93] SunSoft. NIS to NIS+ transition guide. Manual, December 1993.

[Wat81] R. W. Watson. *Identifiers (naming) in distributed systems*, volume 105 of *Lecture Notes in Computer Science (LNCS)*, chapter 9, pages 191–210. Springer, 1981.

[WB94] D. L. Wells and J.A. Blakeley. Distribution and persistence in the open object-oriented database system. In T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, San Mateo, CA, USA, May 1994. Morgan-Kaufmann Publ. Co.

[WW90] I. Williams and M. Wolczko. An object-based memory architecture. In Dearle et al. [DSZ90], pages 114–130.