Vclusters: A Flexible, Fine-Grained Object Clustering Mechanism*

Mark L. McAuliffe
TimesTen Performance Software
2085 Landings Drive
Mountain View, CA 94043
mlm@timesten.com

Michael J. Carey
IBM Almaden Research Center
650 Harry Road, K55/B1
San Jose, CA 95120
carey@almaden.ibm.com

Marvin H. Solomon
Computer Sciences Department
University of Wisconsin—Madison
Madison, WI 53706
solomon@cs.wisc.edu

Abstract

We consider the problem of delivering an effective fine-grained clustering tool to implementors and users of object-oriented database systems. This work emphasizes on-line clustering *mechanisms*, as contrasted with earlier work that concentrates on clustering *policies* (deciding which objects should be near each other). Existing on-line clustering methods can be ineffective and/or difficult to use and may lead to poor space utilization on disk and in the disk block cache, particularly for small- to medium-size groups of objects. We introduce variable-size clusters (*Vclusters*), a fine-grained object clustering architecture that can be used directly or as the target of an automatic clustering algorithm. We describe an implementation of Vclusters in the Shore OODBMS and present experimental results that show that Vclusters significantly outperform other mechanisms commonly found in object database systems (fixed-size clusters and near hints). Vclusters deliver excellent clustering and space utilization with only a modest cost for maintaining clustering during updates.

1 Introduction

Clustering has long been recognized as one of the most effective ways to improve database system performance. The goal of clustering is to place related data objects near one another in physical storage in order to minimize the number of I/O operations needed for common database operations. Good clustering has two components: a *policy* for grouping related objects, and a *mechanism* for storing these groups and maintaining their integrity over time. The design of object clustering policies has been a popular topic for object-oriented database research, but the development of general-purpose clustering mechanisms has received relatively little attention. The *coarse-grained* problem of allocating disk pages (or extents) near one another has been thoroughly studied in the file system community and is considered a solved problem [MJLF84], but the problem of effectively clustering small data objects together, which we will refer to as the *fine-grained object clustering problem*, seems to have received little attention in either the file system or object database communities. This is particularly true for small- to medium-sized groups of objects, and especially in the face of size-changing updates to objects and their clusters.

Despite their popularity as a research topic, clustering algorithms have seen little technology transfer. A large number of automatic clustering algorithms exist in the research literature, but we are aware of none that has been implemented yet in a commercial product. In contrast, most object-oriented database systems, both commercial and experimental, offer some mechanism for clustering. However, the mechanisms provided can be awkward to use and, as we shall show, are not particularly effective. We believe that a first step in delivering effective clustering to commercial database systems is the development of a *general-purpose* clustering mechanism. Therefore, in this research we concentrate on the *mechanism* side of the problem. We assume that the *policy* is given either by hand or as the output of some automatic clustering algorithm.

Our design is directed by the following goals:

Effectiveness: The mechanism should deliver good clustering across a wide variety of workloads.

Low cost: The cost for using and maintaining clusters should be minimal.

Ease of use: Use of the mechanism should be straightforward, and should not require *a priori* knowledge of, or limits on, the sizes of clusters.

Applicability: The mechanism should be usable either as the back-end of an automatic clustering algorithm or directly by application programmers.

Flexibility: The mechanism should support clusters of varying sizes, ranging all the way from fine-grained (less than one page, up to a few pages) to large (multi-megabyte) clusters.

Support for incremental cluster maintenance: The mechanism must gracefully maintain clusters as data items are created or deleted, or as they change in size.

Good space utilization: The mechanism should not waste excessive amounts of disk space.

*This research was sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

The rest of this paper is organized as follows. Section 2 surveys previous work in object clustering and concludes that existing mechanisms fall into two categories: "near hints" and fixed-size clusters. Section 3 introduces a new fine-grained object clustering mechanism that we call *Vclusters*, for variable-sized clusters, and Section 4 describes two algorithms for implementing them. Sections 5 and 6 report on our experiences with Vclusters in the context of Shore: Section 5 describes our implementation of Vclusters in Shore, and Section 6 presents selected results from a performance study comparing the performance of Vclusters with near hints, fixed-size clusters, and no clustering at all. These results show that near hints are scarcely better than no clustering, and that fixed-size clusters impose a penalty in disk-space utilization that adversely affects throughput as well. Vclusters are shown to add a modest overhead to populating and updating the database, but this investment is shown to be quickly repaid through improved performance for clustered traversals. We also show that Vclusters do a better job than the other methods in terms of preserving object clustering across updates. Section 7 summarizes our work.

2 Related Work

The two largest categories of object clustering research are automatic clustering algorithms and special-purpose clustering mechanisms. Due to space limitations, we discuss related research only very briefly here. For a more thorough survey of work in the area of object clustering, we refer the reader to Bertino *et al* [BSI94].

2.1 Automatic Clustering Algorithms

Automatic clustering algorithms focus on policy more than mechanism, and they are therefore only tangentially related to our work. They might be more aptly called *reclustering* algorithms, because their focus is usually on reorganizing an existing object database to match common access patterns. A common approach for these algorithms is to allow database clustering to degrade over time and then to use periodic reclustering to restore performance. In contrast, our work focuses on supporting good initial placement followed by effective incremental maintenance of clusters, in hopes of making such reorganization unnecessary except in the event of changing access patterns. Examples of automatic object clustering algorithms include the work by Hudson and King [HK89], Tsangaris and Naughton [TN91, TN92], McIver and King [MK94], and Gerlhof *et al* [GKKM94].

2.2 Special-Purpose Clustering Mechanisms

Several clustering studies have considered clustering mechanisms that are designed to work in a particular context. Several of these methods are designed to work with specific kinds of data structures. For example, Banerjee *et al* [BKKG88] describe a method for clustering DAGs. As another such example, a recent paper by Zou, Salzberg, and Ladin [ZSL97] describes a similar technique for clustering tree-structured data.

Other special-purpose clustering mechanisms are designed to work with a specific type system or in a specific software context. Chang and Katz [CK89] describe a clustering mechanism that makes use of the structural and inheritance relationships of a CAD data model. Gruber and Amsaleg [GA94] describe a method that works in conjunction with garbage collection. Semantic clustering [SS90] is an interesting technique that combines clustering, object caching, and pointer swizzling. Finally, Cactis [DKH90], a system for managing derived data, uses a special-purpose (off-line) clustering mechanism.

2.3 Other Clustering Research

"Placement trees" [BD89] are a clustering system that was designed in the context of O_2 [Deu90]. Placement trees use programmer-supplied schema annotations to derive a "template" for clustering complex objects. Some network database systems have methods for clustering the members of a parent-child set together, possibly with the set owner [Sal86]. Carey *et al* [CSL+90] describe a similar mechanism, for relational database systems, that was implemented as the Starburst "IMS" attachment [HCL+90]. In all three cases, the initial placement of records is considered, but cluster maintenance is not. Söderlund [Söd81] considers the cost effectiveness of incremental, on-line cluster maintenance in network databases. Cheng and Hurson [CH91] study the same problem in the context of object-oriented databases, although their focus is on large, multi-page clusters (i.e., not on fine-grained clustering). Both studies concluded that such reorganization can indeed be beneficial. Finally, Hornick and Zdonik [HZ87] describe a general-purpose clustering method in the context of the ENCORE OODB. This clustering method is based on multi-page segments, and is similar to the fixed-size clusters described in the next section.

2.4 General-Purpose Clustering Mechanisms

Unlike clustering policies and special-purpose clustering mechanisms, we are aware of no published research in the area of general-purpose clustering mechanisms for database systems. A survey of commercial and experimental OODBs finds only two such mechanisms, near hints and fixed-size clusters.

2.4.1 Near Hints

The most common general-purpose clustering mechanism in object-oriented database systems is *near hints*. A near hint is given as the object identifier (OID) of an existing persistent object. When creating a new object, an application can use a near hint to request that the new object be placed on the same page as the near hint object, if possible. Virtually every object-oriented database system, both commercial and experimental, supports near hints.

While near hints are easy for application programmers to use and for system builders to implement, they have several drawbacks. To begin with, OODBs do not record near hint information persistently in the database, meaning that a near hint can be used only for initial object placement, not for cluster maintenance. Another serious drawback is that there is no way to tell the database system that a given object will be used as a near hint in the future, and that unrelated objects should therefore not be placed near it (to leave space for its related objects). To address this problem, some systems support *fill factors*. A fill factor is typically expressed as a percentage, and it indicates that the system should not place new objects onto data pages that are fuller than the fill factor unless so instructed by a near hint. The use of a fill factor can increase the chances that a near hint will succeed, but it is a rather coarse mechanism and, as our performance study will show, it does little to improve the effectiveness of near hints.

2.4.2 Fixed-Size Clusters

A clustering mechanism that addresses some of the problems with near hints is "fixed-size" clusters, such as those implemented in ObjectStore [Obj95]. A fixed-size cluster is a physical storage structure that is pre-sized to an integral number of pages by the application that creates the cluster. Fixed-size clusters solve the "future near hint" problem mentioned above, leaving space to satisfy future clustering requests, by refusing to place objects from different clusters onto the same page. However, this can lead to severe fragmentation when cluster sizes are not an even multiple of the system page size. An additional drawback to fixed-size clusters is that the eventual size of a cluster may not be known in advance. The size and number of objects in a complex object graph may vary greatly, and the size of an object that contains an embedded collection (set, bag, etc.) may be impossible to guess if the eventual cardinality of the collection or its implementation overheads are not known in advance.

With fixed-size clusters, if an accurate cluster size cannot be calculated in advance, the application programmer must simply guess. If a cluster fills its pre-allocated space, subsequent attempts to add new objects to the cluster will fail. When deciding how the application should handle such failures, the application programmer is faced with few reasonable options. For some complex data structures, it may be possible to spread the structure over a "pool" of clusters, although this method may require modifications to persistent data structures and extra application code to manage the cluster pool. Another option is to maintain a designated "overflow area" for objects that cannot be placed into their preferred cluster, although the use of a single shared overflow area largely defeats the purpose of clustering.

3 A New Fine-Grained Clustering Mechanism: Vclusters

The shortcomings of the clustering mechanisms described above call for a mechanism that is fine-grained, easy to use, provides good clustering and space utilization, and facilitates both smart initial object placement and incremental cluster maintenance. To fill this need, we propose a new fine-grained object clustering mechanism based on variable-size clusters, or *Vclusters*. As we will show later, Vclusters are as easy to use as near hints, yet they are capable of providing the same quality of clustering as fixed-size clusters without their inherent fragmentation problems. Vclusters are physical storage containers that can grow and shrink in size as objects are added and removed. They are designed to support a wide range of cluster sizes efficiently, ranging from fine-grained clusters of hundreds of bytes up to large clusters of megabytes or more. Vclusters are also designed to function as "sticky near hints"; information about clustering is stored persistently in the database and can therefore be used for both initial object placement and incremental cluster maintenance.

Vclusters can be used in one of two ways. They can be used directly by application programmers, much like near hints and fixed-size clusters are currently used, or they can be used as the back-end of an automatic clustering policy. In the case of direct use by application programmers, we believe that Vclusters' support for variable-sized clusters is essential to achieve good performance in combination with ease of use. In the back-end case, variable-sized clusters can still be very useful. For example, the placement tree approach [BD89] uses programmer-supplied annotations to derive "logical clusters," which can be of any size, and then maps these logical clusters onto fixed-size pages. The designers of placement trees used near hints to perform this mapping, but Vclusters would have been a much better back-end for placement trees due to the strong similarity between Vclusters and logical clusters. Similarly, Vclusters would be an excellent fit for addressing problems such as incrementally maintaining parent-child set clustering in network database systems or in the IMS attachment of the Starburst system (see Section 2). Basically, the Vcluster mechanism is directly applicable to the cluster creation and maintenance problems of any clustering approach that identifies logical clusters as a step in the object clustering process. In contrast, certain clustering policies, such as those of Tsangaris and Naughton [TN91, TN92], directly compute an exact desired grouping of objects together into individual disk pages. Policies such as these, which go directly from access traces to pages without identifying logical groupings of objects, would not take advantage of Vclusters' flexible support for cluster size variability; however, the "stickiness" of the Vcluster mechanism would still be helpful in maintaining the prescribed clustering over time.

The interface to Vclusters looks much like near hints to the application programmer: To create an object in a specific Vcluster, the application provides the OID of an object that already resides in the cluster. An application may also choose to *open* a Vcluster if it plans to create several new objects in it. The *open* operation accepts the OID of an existing object and returns a transient *cluster descriptor* for the cluster that contains the object. Cluster descriptors are analogous to the file descriptors bound to open files in operating systems. In addition to providing a convenient way to name the cluster in subsequent object-creation operations, the *open* operation (and the corresponding *close* operation) serves as a hint to the database system that the cluster will be growing—information that the system can use to optimize performance. As a further aid to the implementation, the *open* operation accepts an optional "size hint" indicating the application's best guess as to the amount of data that will be added to the cluster while it is open. (We emphasize that this is strictly an *optional* size hint; we will see later that it can be helpful, but is by no means necessary, to the operation of Vclusters.)

3.1 Layout Issues

The algorithms and heuristics to be presented in Section 4 try to keep Vclusters close to their "ideal" layout on disk. Before we consider these algorithms, however, we must answer the question of what constitutes an "ideal" layout. For example, consider an application in which each cluster is precisely 2/3 of a page in size. Two possible layouts for this application are a "sparse" layout, in which each cluster lives on a page by itself, and a "dense" layout, which packs three clusters into every two pages, splitting one-third of the clusters (see Figure 1). The dense layout is clearly preferable from the point of view of space utilization (100% *vs.* 67%), but which layout gives better runtime performance for a highly clustered database traversal?

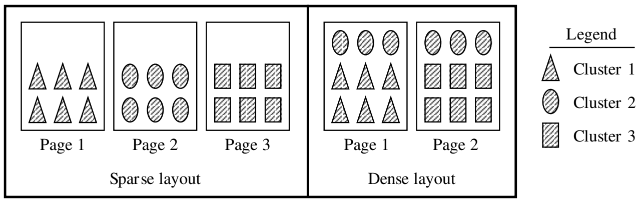

Figure 1: Sparse and dense cluster layouts.

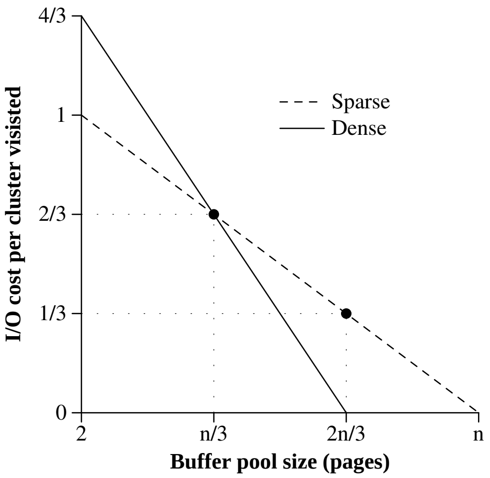

Figure 2: Predicted I/O cost for traversals of sparse and dense layouts.

The answer depends on the size of the buffer pool and the nature of the access pattern. Consider a traversal that repeatedly chooses a random cluster and visits every object in that cluster. If the buffer pool is very small, each cluster must be read into memory before it is traversed. The average cost to read a cluster is one I/O for the sparse layout and 4/3 I/Os for the dense layout (since unsplit clusters require one I/O, while split clusters require two). On the other hand, if the database contains n clusters, a buffer pool of $(2/3)n$ pages can hold the entire database with the dense layout (so the I/O cost drops to zero in the steady state), but the same buffer pool can hold only 2/3 of the database for the sparse layout (yielding an expected cost of 1/3 I/O per cluster). The expected cost, in I/O operations per cluster, is a linear function of the buffer pool size, as shown in Figure 2. A sparse layout is preferable if the buffer pool holds fewer than $n/3$ pages, while a dense layout gives better performance for larger buffer pools.

Although this simple analysis ignores such issues as variations in cluster sizes, complex patterns of locality, and CPU costs, it shows that in some cases, the best traversal performance is obtained by a policy that minimizes the number of split clusters, while in other cases, the best performance is obtained by a policy that optimizes space utilization. However, it is important to note that although the dense layout is never more than a bounded factor worse than that sparse layout, it may be better by an arbitrary factor. For example, in Figure 2), the dense layout is 33% worse than the sparse layout when the buffer pool holds two pages, but for a buffer pool exceeding $2n/3$ pages, the sparse layout is worse by an infinite factor (a non-zero cost divided by a zero cost)

For medium-sized clusters (up to a few pages in size), we are faced with a similar choice. In a sparse layout, each cluster is allocated as many pages as necessary to hold it. In a dense layout, some pages are "mixed"—they contain fragments of multiple clusters. The sparse layout can waste up to half of the disk (e.g., if each cluster is just slightly more than one page in size). On the other hand, the dense layout is much harder to maintain as clusters grow and shrink. The space penalty (and corresponding I/O performance penalty) for the sparse layout becomes less significant as clusters get larger, becoming vanishingly small for multi-megabyte clusters. The greatest challenge, therefore, is to manage smaller clusters of objects efficiently. Finally, we should note that our approach intentionally treats clusters as logical units of access, as we assume that all of the objects in a cluster have similar co-access affinities. Thus, our approach to the clustering problem is "flat"—our implementation supports files of objects partitioned into clusters—as opposed to a multiple level clustering model. This is clearly appropriate for fine-grained clustering, and we believe that relatively little additional performance benefit would be gained from a more complex model supporting nested or overlapping clusters.

These considerations led us to an implementation of Vclusters with three representations of clusters and two kinds of pages. Allocated pages are designated as *mixed* or *dedicated*. A mixed page may hold objects from multiple clusters[1], while a dedicated page may hold objects from just one cluster. Clusters are designated as *small*, *medium*, or *large*. A small cluster is less than a page in size, residing on a portion of one mixed page; large clusters consist entirely of dedicated pages, while medium clusters can include a combination of dedicated pages and portions of mixed pages. A cluster that grows beyond a certain size is promoted to large by moving objects off of mixed pages onto newly allocated dedicated pages. A large cluster that gets too small is demoted, allowing it to share pages with other clusters. To prevent clusters from "bouncing" back and forth from the large representation, we use two size thresholds. In our Shore implementation, clusters become designated as large when they exceed twelve pages, and they are reclassified as medium when they drop below eight pages. These thresholds are somewhat arbitrary, but seem to work well in our experiments. The lower threshold was chosen to ensure space utilization of at least 87% (7/8) for large clusters even when one page is nearly empty. The higher threshold was chosen to be comfortably larger in order to provide hysteresis as protection against "bouncing".

4 Vcluster Layout Algorithms

Having discussed the tradeoffs involved in mapping the objects in Vclusters to disk pages, we now consider algorithms for performing these mappings. Prompted by the analysis of Section 3.1, we have developed two Vcluster algorithms: *Vcl-dense*, which tries to maintain good space utilization at all times, even if it is necessary to split clusters to do so, and *Vcl-minsplit*, which tries to minimize cluster splitting, even

[1] Our mixed pages are similar in concept to fragments in the 4.2 BSD fast filesystem [MJLF84].

if space utilization suffers as a result. Although each algorithm has a distinct bias, both algorithms try to minimize cluster splits and maximize space utilization when possible.

4.1 Free Space Management

Both of our Vcluster algorithms are based upon an earlier algorithm that we developed for allocating and freeing objects [MCS96]. We briefly review that space management algorithm here; the reader is referred to the full paper for further details.

We assume that objects are allocated from a *file* of *pages*. As objects are deleted, pages become partially emptied. When a new object is to be allocated, storage allocation algorithms in existing database systems use one of two approaches. They may search all of the pages of the file for one with enough free space (an "exhaustive-search algorithm") or allocate a new page and add it to the file (an "append-only" algorithm). Exhaustive search can leads $O(n^2)$ performance for adding n objects to a file. Append-only leads to poor space utilization: If free space on a page is not reused until the page becomes empty, space utilization can be expected to approach $1/H_k$ in the limit, where k is the number of objects that fit on a page and H_k is the k^{th} harmonic number. For example, for 256-byte objects allocated from 8k-byte pages, utilization drops to $1/H_{32} \approx 25\%$, and for 32-byte objects, it drops to $1/H_{256} \approx 16\%$ (see [MCS96] for a proof). Many commercial systems provide a reorganization utility that consolidates under-filled file pages, releasing any pages that become completely empty in the consolidation process. However, these utilities tend to be highly disruptive to normal system operation.

Our solution to this problem was a hybrid of exhaustive search and append-only called HY(n,u). The algorithm maintains an estimate of the current space utilization of the file (the sum of the sizes of the objects in the file divided by the sum of the sizes of the pages). If that utilization drops below u (a parameter of the algorithm), the allocator uses a form of exhaustive search to allocate new objects. Otherwise, it only searches the n most recently used pages. In all experiments described in this paper, $u = 7/8$ and $n = 8$.

To keep track of free space, we use one persistent data structure and two transient data structures created when a file is opened. The persistent structure is a *space map*, an array of 4-bit entries, one for each page of a file. Each entry indicates the amount of free space on a page, classifying it into one of 16 *free-space classes*.[2] The transient structures are a 16-entry *free-space histogram*, which counts the number of pages of a file in each free-space class, and a *pid cache*, which records the page identifier and free-space class of each of n recently used pages. The histogram is easily derived from the contents of the space map. The algorithm also maintains an estimate of the current space utilization, computed from the free-space histogram.

When creating a new object in a given file, HY(n,u) always attempts to allocate the object on a page whose id is listed in the pid cache. If no page listed in the cache can accommodate the object, the algorithm will either search the file's space map (using a "next-fit" strategy) to find an existing page that can hold the object, or allocate a new page to the file. It will search for an existing page only if the overall space utilization of the file is currently below u and the free-space histogram indicates that there is at least one page in the file that can hold the new object. For simplicity, we do not concern ourselves here with the problem of allocating the individual pages of a Vcluster close to one another; as mentioned earlier, that is a relatively well-understood problem in the operating systems community, e.g., see [MJLF84], and it was also addressed in the work of Cheng and Hurson [CH91]. Whatever page is used to allocate the new object is added the pid cache, replacing the pid cache entry for the fullest page in the cache. Our experiments indicate that a small pid cache (containing just 8 entries) is adequate to maintain a high utilization (87%) with low overhead over a wide variety of workloads, even for very large files [MCS96].

4.2 The Vcl-minsplit and Vcl-dense Algorithms

To create and delete new objects in large clusters (which consist entirely of dedicated pages), both algorithms simply use the HY(n,u) algorithm within the dedicated pages of the target cluster. Henceforth in this section, we will omit mention of large clusters, focusing instead on how the algorithms treat small and medium clusters.

We use the term *fragment* for the portion of a cluster that resides on any one page. Each cluster consists of a set of dedicated pages plus a *sequence* of fragments on mixed pages. In a slight abuse of terminology, we shall refer to these latter fragments as "mixed fragments." Vcl-minsplit maintains the invariant that every cluster has zero or more dedicated pages plus at most one mixed page. In contrast, (non-large) Vcl-dense clusters have *only* mixed pages; in other words, Vcl-dense does not use dedicated pages for its medium clusters. The fragments of a Vcl-dense small cluster are numbered in order of creation, starting at zero. To control the number of fragments, Vcl-dense attempts to keep the size of fragment i above $\min(2^i \cdot \mathrm{MST},\ \mathit{pagesize})$, where MST, the *minimum size threshold*, is a constant (1/8 page in our implementation). This limit prevents Vcl-dense from spreading a cluster's objects over an excessive number of fragments; they start out small, but Vcl-dense tries to make each successive fragment at least twice as big as the previous one. The highest numbered mixed fragment is called the *active* fragment, and its page is termed an *active* page. A fragment is said to *outgrow* its page when that page does not have enough free space to accommodate an object creation request. The principal difference between the two Vcluster algorithms is how they cope with outgrowing a page. Vcl-minsplit may move the fragment to an emptier page, while Vcl-dense may decide to leave the fragment where it is and begin a new fragment on another mixed page.

Figures 3 and 4 on the next page illustrate these ideas. In both figures, Cluster 1 is a large cluster containing nine objects stored on dedicated pages 1–3. In Figure 3, Cluster 2 is a medium cluster stored on dedicated pages 4 and 5 and mixed page 8. Medium cluster 3 also used part of mixed page 8, as well as dedicated page 6. Small cluster 4 is on page 9, while small clusters 5 and 6 share page 7.

In Figure 4, Cluster 2[3] is split into four fragments stored

[2] This low-precision representation of the amount of free space not only allows a more compact space-map, it decreases the frequency of changes to entries, an important consideration in a database system that must log all changes to persistent data structures to support crash recovery.

[3] There is no distinction between medium and small clusters with the Vcl-dense algorithm.

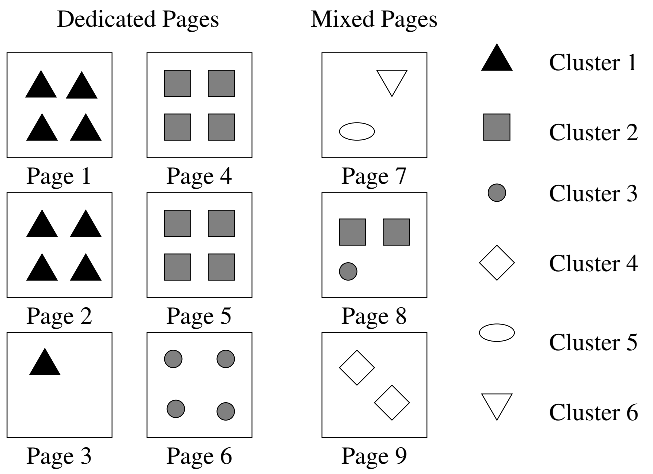

Figure 3: A typical layout of clusters with the Vcl-minsplit algorithm.

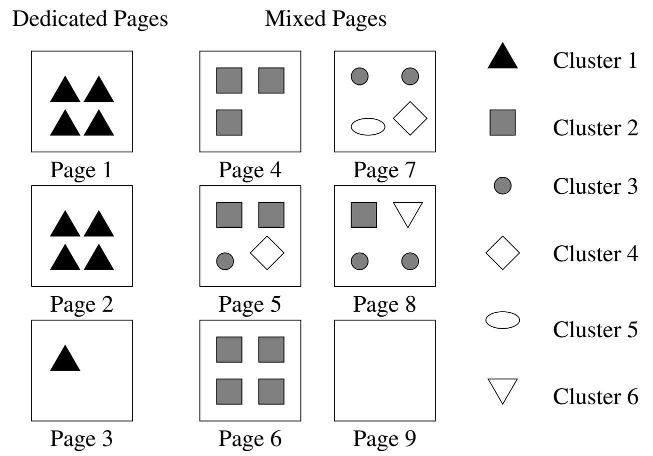

Figure 4: The objects of Figure 3 as laid out by the Vcl-dense algorithm.

on pages 8, 5, 6, and 4 (in that order). The active page for cluster 2 is page 4. Similarly, cluster 3 has fragments on pages 5, 7, and 8, while cluster 4 is on pages 5 and 7.

Both algorithms take advantage of the *close* operation to reorganize a cluster, thereby improving space utilization and/or decreasing the number of fragments. (Vcl-dense tries harder to do the former, while Vcl-minsplit optimizes for the latter.) Unfortunately, a shrinking cluster, unlike a growing cluster, is unlikely to be open, as an application may know little or nothing *a priori* about the cluster membership of objects that it is deleting. We therefore use a heuristic to detect shrinking clusters and reorganize them when they stabilize. With each transaction, we associate a fixed-size LRU list of clusters from which the transaction has deleted objects and that require further attention. (This list is simple to maintain since each object on disk is tagged with its cluster id in our implementation). When the deletion of an object from a page causes the page's utilization to drop below u, the cluster that contained the deleted object is added to the LRU list (or is moved to its head if it was already there). When a cluster is ejected from the LRU list, it is reorganized as if it were the subject of a *close* operation. Clusters remaining in the LRU list when the transaction commits are reorganized at that time to ensure that the reorganization will occur even in the event of a post-commit system crash.

In somewhat more detail, the algorithms work as follows:

Allocating a mixed page

When either algorithm needs to allocate a mixed page for creating or copying a fragment, it allocates a new, empty mixed page unless doing so would cause the overall space utilization of the file to drop below the target u. It also refuses to use the active page of another cluster or a page with insufficient free space. Specifically, a new fragment requires a page with at least MST bytes of free space, and a fragment being moved from another mixed page requires an amount of free space that is at least twice the current size of the fragment. These restrictions attempt to limit the number of times a growing segment will end up being recopied. Because a fragment is active, it "stakes a claim" on the free space on its page, and the doubling of the claim on each copy ensures that the total number of bytes copied is less than twice the ultimate size of the cluster.

Creating a new object

In the Vcl-minsplit algorithm, new objects are added to dedicated pages of medium clusters using HY(n,u) (dedicated pages are only used for large clusters in Vcl-dense). Otherwise, both algorithms attempt to allocate them on the active page of the cluster.

Outgrowing a page

This case occurs when an attempt to add a new object to the active page of a small cluster fails because there is not enough free space on the page.

Vcl-minsplit: If the active page contains more than one cluster, the active fragment is copied to another mixed page, chosen as above. Otherwise, the cluster is promoted to medium status and is given a dedicated page which is used to allocate the new object.

Vcl-dense: If the active fragment is less than its "minimum" size ($\min(2^i \cdot \mathrm{MST},\ \mathit{pagesize})$), or if the file contains a mixed page with an amount of free space that is at least twice the size of the active fragment, then the active fragment is copied to a new mixed page (chosen as above). Otherwise, a new fragment is added to the list of fragments, allocated to a mixed page (again, chosen as above), and designated to be the cluster's active fragment and used to allocate the new object.

Reorganizing a cluster

When an open cluster is closed, or when a shrinking cluster is "closed" because it is expelled from a transaction's LRU list of shrinking clusters, it is reorganized.

Vcl-minsplit: If the cluster is small, occupying some fraction of a single mixed page, there is nothing to do. Otherwise, the reorganization proceeds in two phases.

In the first phase, Vcl-minsplit coalesces free space on the cluster's dedicated pages by moving objects onto fuller dedicated pages from its emptier dedicated pages and mixed page. Any pages that become empty during this phase are freed. If, at the completion of this phase, there are any remaining "under-full" dedicated pages (ones with utilization less than u), the second phase combines the cluster fragment on the emptiest of the under-full pages with the cluster fragment on the mixed page (if any), placing this new cluster fragment onto the tightest-fitting mixed page in the file, allocating a new mixed page for it if needed.

Vcl-dense: Any under-full pages of the cluster are arranged into a list ordered by fullness. They are then compacted by moving objects from emptier to fuller pages of the cluster, starting at the ends of the list and moving towards the middle as pages fill or as cluster fragments have been completely moved. If an under-full page remains and its fragment can be moved to another mixed page whose utilization would thereby be increased beyond u, the fragment is moved.

4.3 Size Hints

One of the primary benefits of Vclusters over fixed-size clusters is that the creator of a Vcluster is not required to supply the size of the cluster at the time of its creation. However, in some cases, an application may in fact know the size of a cluster in advance, or may at least be able to supply an estimate of the cluster's expected size. Such information, if accurate, could be used by our Vcluster algorithms to improve performance by avoiding some copying of fragments. Our Vcluster algorithms use these optional size hints, when provided, as follows:

Vcl-minsplit: When creating the first object in a new cluster, Vcl-minsplit considers the cluster to be large or medium-sized from the outset if the size hint indicates that the cluster will be larger than a page. It then places the cluster onto a dedicated page. If the size hint indicates that the cluster will be smaller than a page, then a mixed page is chosen that can hold an entire cluster of that size, even if a new page must be allocated to do so.

Vcl-dense: Vcl-dense initially places the cluster onto an empty mixed page if the size hint indicates that the cluster will be larger than one page, which usually requires that a new page be allocated. If the size hint indicates that the cluster will be smaller than one page, then the cluster will be placed onto a mixed page that can hold an entire cluster of the predicted size, if such a page exists. However, unlike Vcl-minsplit, Vcl-dense will only allocate a new mixed page for the cluster in this case if there is no mixed page that has at least MST free bytes.

5 Implementing Clustering in Shore

Section 6 presents a performance study based on actual implementations of Vclusters, fixed-size clusters, and near hints in Shore. We modified Shore's *file* data structure, which formerly supported only a flat set of pages, to consist of a collection of lists of pages. To implement Vclusters, we used one page list in each file to keep track of all of the mixed pages in the file, and we used separate page lists to keep track of the set of dedicated pages for each medium or large cluster in the file. HY(n,u) was used to manage space within each individual page list. The header of each Shore data object was extended to include a cluster id. Finally, each Shore volume also had an associated persistent B$^+$tree for mapping <file, cluster id> pairs to the associated list of mixed pages to make each cluster's mixed page list readily accessible.

To implement fixed-sized clusters, each cluster was represented as a page list whose pages are all allocated when the cluster is initially created. For the near hints algorithm (and our baseline "no clustering" algorithm), each file had only one page list. When creating an object in the presence of a near hint, our implementation allocates the new object on the page that contains the indicated object if possible. Otherwise, it uses HY(n,u) to place the object elsewhere in the file. Further details of our modifications to Shore, as well as of our implementations of the different clustering algorithms, can be found in [McA97].

5.1 Issues in Moving Objects

Our Vcluster algorithms depend on the ability to move objects from one page to another. Shore supports object movement with logical OIDs implemented by an OID index and an LRU cache of logical-to-physical translations (the "OID cache"). Two issues related to transactions complicate the task of object movement. The first concerns lock conflicts that arise when an object to be moved for cluster-maintenance reasons is locked by another transaction. Our current implementation simply waits for the lock; this is adequate for the tests described here, but could lead to poor concurrency and occasional deadlocks in a production system. More thorough solutions are possible, e.g., based on special-purpose locking and logging protocols to allow clustering-related moves of locked objects, or based on the maintenance of a persistent "to-do" list of the objects that need to be moved later on and a background thread that processes this list. Experimentation with such high-concurrency solutions is beyond the scope of the current study.

The second issue concerns what happens when a transaction aborts after moving one or more objects for clustering reasons. Our implementation logs the moves as ordinary operations in the context of the transaction that caused them, which means that if the transaction aborts, its moves will be undone. Because of this, when a transaction moves an object, it must reserve the space on the old page in order to ensure that it can move the object back to its original location in the event of a rollback. Furthermore, the transaction must retain a lock on the object until it commits in order to prevent other transactions from increasing the size of the object, as a larger object might not fit on the old page, were it to be moved back. However, it is safe for the transaction to consume any reserved space that it may itself have on the new page when placing the object there. An alternative to this implementation would have been to enclose object movements inside *nested top-actions* [MHL$^+$92], which are subsequences of the actions of a transaction that are not undone even if the enclosing transaction aborts. Nested top-actions are typically used when a transaction makes changes to persistent structures, such as splitting a B$^+$tree page, that have no semantic effect on the database but that do affect other active transactions. An examination of the relative benefits and drawbacks of using nested top-actions for object movement is also beyond the scope of this paper.

6 Performance Results

To study the costs and benefits of the clustering mechanisms described in this study, we constructed a set of experiments that measure the performance characteristics of cluster creation, traversal, and maintenance for each of our algorithms and compared the measured results for Vclusters with implementations of fixed-size clusters and near hints with a fill factor. We also tested the performance of a baseline "no-clustering" option, for which we simply used the HY(n,u) algorithm with no clustering extensions. Table 1 summarizes the clustering algorithms and the labels that we will use to identify them when describing the results. We considered the possibility of using an existing OODB benchmark such as OO7 [CDN93] for these experiments; however, benchmarks like OO7 have placed too little emphasis on clustering-related operations to be effective for this purpose [CDK+94], and the planned extensions to OO7 in this area were never completed.

After exploring the design space for clustering performance studies for quite a while, we settled on a set of five experiments for use in evaluating clustering algorithm performance. The first experiment measures the cost of building a database, while the second and third evaluate the quality of clustering achieved by measuring the cost of two kinds of clustered traversals. The fourth experiment applies a series of updates to the database, creating and deleting random clusters and objects, and measures both the performance of the updates themselves and the effects of the updates on subsequent read-only traversals. Together, this set of experiments subjects the set of clustering algorithms to a variety of important OODB workloads and operating conditions; both loading and incremental update conditions are covered, and several varieties of traversals are considered. The final experiment measures the effects of both accurate and inaccurate size hints on the performance of the two Vcluster algorithms.

Algorithm	*Label*
Vcl-dense	VDENSE
Vcl-minsplit	VMIN
Fixed-size clusters	FIXED
Near hints with 87% fill factor	NEAR87
No clustering	NONE

Table 1: Clustering algorithms and their labels.

The hardware platform for our experiments consisted of a Digital Equipment Corporation DECpc XL 590 (90 Mhz Pentium processor) with 32 megabytes of main memory; the operating system used was Solaris 2.4. The data volume was a two-gigabyte Seagate ST12400N hard disk that was mounted as a raw device entirely under the control of Shore. The log was stored on a 100-megabyte raw partition of a Conner CFP 1060S hard disk. To avoid interference from outside sources, the machine was removed from all networks, and all unnecessary processes were terminated.

The parameters underlying the performance results shown here are summarized in Table 2. To keep the experimentation time manageable, the database size chosen for our experiments was fairly small (approximately 32 MB); to balance the database size and main memory size appropriately, we used a correspondingly small buffer pool (512 8 KB pages, totaling 4 MB). On the other hand, we used a relatively large OID cache—3 MB—and we initialized this cache before each traversal experiment to hold all OID translations in the database. This was done not to mimic a realistic setting, but rather to minimize the (potentially distracting) effects of logical ID lookups on our traversal results; our goal in doing this was to obtain less Shore-specific results by factoring out the performance effects of Shore's particular OID translation mechanism. All of our clustering methods are based on HY(8,87) (i.e., HY with an 8-entry pid cache and a target utilization of 7/8, which as mentioned earlier was previously shown to be very effective [MCS96]). For Vcl-minsplit and Vcl-dense, the minimum size threshold (MST) was set to one-eighth of the system page size. Finally, for near hints, a default fill factor of 87% was used to achieve a balance between leaving too much and too little free space; we also explored alternative fill factor settings in one of our experiments (which will be discussed shortly).

Parameter	*Value*
Page size	8 Kbytes
Buffer pool size	512 pages
OID cache size	3 Mbytes
Page cache size (n)	8
Target space utilization (u)	87%
Minimum size threshold (MST)	1 Kbyte
Interleaving factor	8
Fill factor (near hints only)	87%
Root node size	428 bytes
Leaf node size	200–800 bytes
Number of leaves/tree	See Figure 6

Table 2: Parameters used in most of the clustering experiments.

Our test data itself was designed to provide the algorithms with a database consisting of groups of related objects that tend to be accessed together. We wanted this database to be easy to generate in a controlled way, and to provide an easy means to vary the size of these groups of objects in our experiments. In general, the problem of deciding how to group application objects into clusters is a challenging problem, but it is outside the scope of this work. We assume that the application designer or one of the cluster-defining algorithms cited earlier has grouped objects into clusters in such a manner that "clustered traversals"—traversals in which references to members of a cluster have high temporal locality—are common.

To these ends, we constructed a test database of two-level trees of objects (see Figure 5). (We experimented with several more complicated structures as well, but we gained no additional insights from their added complexity.) Each of our two-level trees has up to 48 leaf objects and a root object large enough to hold 48 object identifiers (428 bytes including overhead). The size of a leaf object is a random variable uniformly distributed between 200 and 800 bytes.

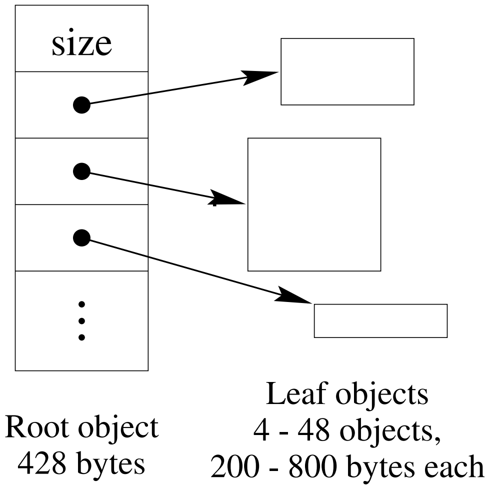

Figure 5: A typical tree in the test database.

The number of leaf nodes also varies randomly from tree to tree. Except as noted below, the number of leaves in each tree is drawn from the distribution $3X + 4$, where X is a Gamma random variable with density function $f(x) = xe^{-x}$ (see Figure 6). As the graph indicates, this distribution gives most trees between four and 15 leaves. In more detail, the distribution has a minimum of 4, a mean of 10, a standard deviation of $3\sqrt{2}$, and an "effective" maximum of 48: The probability that the number of leaves is greater than 48 (the maximum number of leaf nodes the data structure can accommodate) is less than 10^{-5}; any such values were simply discarded. The resulting trees are between 30% and 300% of the system page size, with a mean of 67%, providing quite a wide range of cluster sizes for testing the abilities of different clustering algorithms to manage fine-grained object clusters.

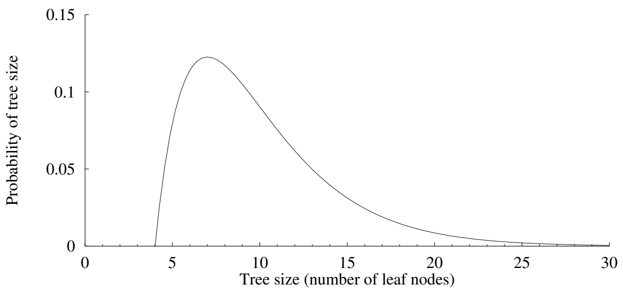

Figure 6: Distribution of tree sizes.

For our test workloads, as described above, we wanted sets of accesses and updates that would test the algorithms' ability to expedite clustered traversals and maintain good performance in the face of updates. We thus created workloads from combinations of five operations: *create*, *destroy*, *grow*, *shrink*, and *visit*. The *create* operation creates a cluster of objects by creating a new tree and populating it with a number of leaf nodes determined by the above Gamma distribution. The remaining operations each choose a random cluster, i.e., an existing tree, by selecting uniformly from among all of the trees in the database. The *destroy* operation destroys an object cluster by destroying all of its tree's leaves and then destroying its root. The *grow* and *shrink* operations change the size of a cluster by adding or removing some number of leaves from the cluster's tree structure. The *visit* operation accesses all of the objects in a cluster by visiting all of its tree's leaves in random order.

In the case of the *grow* and *shrink* operations, the number of leaves added to or deleted from the tree is chosen uniformly from the range 4–8. The *shrink* operation can cause underflow if the number of leaves chosen for deletion is greater than the number of leaves in the chosen tree. This situation is handled by first removing all of the leaves (but not the root) from the tree and then choosing another tree upon which to continue the operation. A similar process is used to handle tree overflows (as root nodes can hold only 48 leaf pointers each). In all tests, overflow and underflow occur so infrequently as to have no significant effect on our results.

For Vclusters, we placed each tree of objects into a separate cluster. For near hints, the root node of each tree was created with no near hint, and each leaf node was created using the root of its tree as a near hint. Fixed-size clusters complicated our "application program" considerably. Since the size of a tree is not known in advance and trees vary in size from one-third of a page to three pages, it would be impractical to make the fixed cluster-size big enough to accommodate the largest possible tree, and we needed a way to recover when a tree "overflowed" a "typical" cluster. Therefore, we used multiple clusters for each tree and added to the root node a list of cluster ids and an indication of the "current" cluster. When adding a node to a tree, the program tries to add it to each of its clusters, starting with the current one. If they are all full, it adds a new cluster and makes it current. Note that our implementation of fixed-size clusters does not need to read pages from disk to determine whether a cluster has room for an object-creation request, as this information is available from the space map. We chose a cluster size of one page (8 KB) because an average tree is 2/3 of a page in size.

Because a system based on near hints (or "no clustering") does not attempt to separate growing clusters from one another, it may not be able to maintain good clustering in the presence of concurrently-growing clusters. To study the effects of multiple concurrently growing and shrinking clusters, we introduced a workload parameter called the *interleaving factor* and created a table containing that many active "operations." The main body of the test program is a loop that repeatedly chooses one of the active operations from the table and performs the next step (create, delete, or visit a leaf node) in that operation. This use of interleaving captures the storage management impact of having concurrent transactions working on one file; it also models what would happen with a single transaction concurrently updating multiple clusters within one file. The interleaving factor was eight in all experiments except as noted.

Finally, to simulate applications that operate on different portions of a database at different times, i.e., to model temporal locality, our test program places each tree in the database into one of eight separate files when it is created. Varying the number of files from which trees are selected during database traversals enables us to model working sets of different sizes.

It is important to note that, while our synthetic database consists of two-level trees, and our workload is therefore made up of tree operations, our results are not specific to trees. This database and workload combination was simply used as a convenient way to create and operate upon a database containing a number of fine-grained clusters containing various numbers of objects, themselves of varying sizes, in interesting ways that would allow us to explore the database creation, incremental update, and access performance characteristics of our clustering algorithms in a well-controlled manner.

6.1 Experiment 1: The Create Workload

Our first experiment measures the extra cost required to initially build a database using Vclusters as compared to the simpler clustering mechanisms. The workload consists entirely of *create* operations. The workload creates 6,400 clusters, spread uniformly over eight initially-empty files, for a total of approximately 32 Mbytes of data.

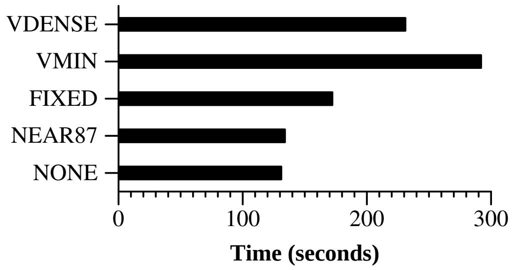

Figure 7: Create workload: Elapsed time.

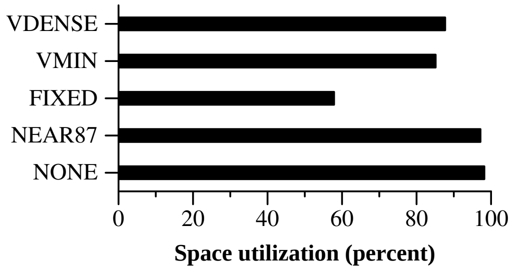

Figure 8: Create workload: Space utilization.

Figure 7 shows the elapsed time required to build the database for each of the tested clustering methods, and Figure 8 shows the resulting space utilization. (Table 1 lists the abbreviations for all of the clustering algorithms tested here.) The three methods that actually create clusters (VDENSE, VMIN and FIXED) are clearly slower than the simpler methods (NEAR87 and NONE), with VMIN taking twice as long to build the database as NEAR87 or NONE. In the case of the Vcluster algorithms, the extra time stems almost entirely from the cost of moving objects from page to page as clusters grow. Moving objects generates additional log traffic, requires updates to the OID index, and can also result in extra page reads and writes. Thus, it is not surprising that Vclusters are slower than the other methods. FIXED is slightly slower than NEAR87 and NONE because it produces a much worse space utilization than the other methods do (just 58%) and therefore performs more page writes during the database building process.

The space utilization results shown in Figure 8 show that VDENSE achieves its target space utilization of 87% and VMIN comes close (85%). If each cluster were exactly 67% of the page size, then VMIN's space utilization would also be exactly 67% because it refuses to split any small clusters. However, because it exploits the variation in cluster sizes by placing clusters (or cluster fragments) of different sizes together on the same page, it yields much better utilization. In contrast, FIXED gets only 58% utilization. Although the average tree uses 2/3 of a page, trees that are just over one page in size waste most of a second page, driving the average utilization even lower. NONE and NEAR87 achieve over 97% utilization since they are at liberty to pack pages at the object, rather than cluster, granularity.

6.2 Experiment 2: Uniform Traversals

Although Experiment 1 shows that Vclusters are slower than the simpler methods for create-only workloads, Experiments 2 and 3 show that the cost is quickly recouped by improvements in the performance of clustered traversals. These experiments use the database created by Experiment 1 and visit each cluster an average of four times.

Experiment 2 repeatedly chooses a random cluster selected uniformly from the database (with replacement) and traverses all of the nodes of its tree, repeating 25,600 times. Figure 9 shows the elapsed time and number of page reads for each algorithm. These graphs indicate that I/O cost is virtually the dominating factor determining the traversal times, particularly in the less efficient cases. All of the other traversals are similarly I/O bound; in subsequent cases where the I/O graphs closely match the elapsed-time graphs, we will show only the elapsed-time graphs.

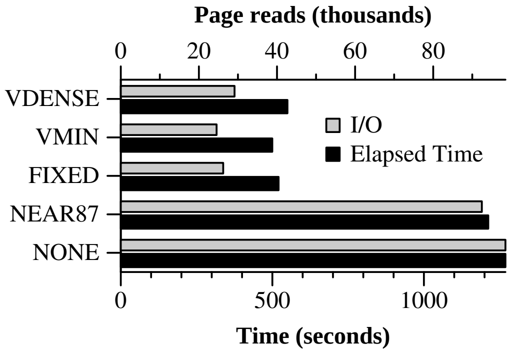

Figure 9: Uniform traversal.

As the analytical model of Figure 2 predicts, VDENSE is marginally worse than VMIN for a relatively small buffer pool. The difference is less marked than the model predicts because we are only requiring VDENSE to achieve 87% space utilization rather than 100%, and because the variation in cluster sizes allows VDENSE to reach that utilization by splitting fewer clusters than would be required if all clusters were exactly 67% of a page.

The most striking results of this experiment are

- Near hints have virtually no effect. NEAR87 is only 5% faster than NONE.
- The higher cost of Vclusters for creating the database in Experiment 1 is quickly repaid by improvements in clustered traversals. The traversals of the Vcluster databases are nearly 2.5 times as fast as the ones built

with near hints or no clustering. In particular, the extra 100 seconds or so to build the database was paid back after visiting each cluster on the average only slightly more than one time.

- Fixed clusters, despite their poor space utilization, work about as well as Vclusters in speed of traversal, because the poor utilization has a small relative effect on the buffer-pool miss rate. The VMIN database is 10 times the buffer-pool size, for a miss rate of about 90%, while the FIXED database is 14 times times the buffer-pool size, for a miss rate of 93%, yielding only a 3% difference.

Effects of the Interleaving Factor

The performance of NEAR87 and NONE for the Uniform traversals shown in Figure 9 suggests that near hints are ineffective at clustering objects when there are multiple clusters growing at once. This hypothesis is verified by Figure 10, which shows the results of re-running Experiment 2 on databases constructed without any interleaving (i.e., with an interleaving factor of one.) For comparison, the elapsed time data from Figure 9 is repeated in Figure 10. This graph shows clearly that the inability of NEAR87 and NONE to separate growing clusters accounts for their poor performance in Experiment 2.

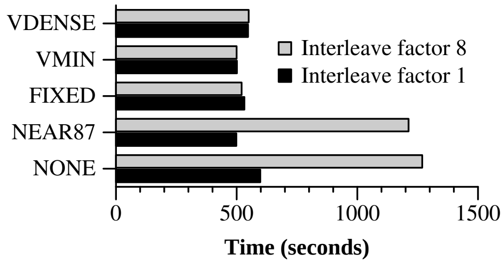

Figure 10: Uniform traversal (with and without interleaving): Elapsed time.

Effects of Fill Factors

A *fill factor* provides a method to try to improve the effectiveness of near hints in the presence of interleaving. The idea behind fill factors is to set aside some amount of space on a page for objects created without near hints to allow clusters to grow around them. To examine the effects of fill factors, we ran the Uniform traversals over databases built with near hints using fill factors of 100%, 75%, and 50% (all with an interleaving factor of 8). The elapsed time results for these traversals are shown in Figure 11, along with the previous results for near hints with an 87% fill factor and for Vcl-dense (both copied from Figure 9).

Although the use of fill factors in conjunction with near hints clearly has some effect on traversal performance, this effect is minimal—even with a fill factor of 50%, the traversal is nearly twice as slow with near hints as it is with VDENSE. The reason for the ineffectiveness of fill factors is that they do not address the basic problem with near hints: Near hints allow objects from too many different clusters to be placed onto the same page, resulting in many split clusters. Fill factors only prevent pages from becoming completely full of objects that belong to different clusters. Their effectiveness is therefore minimal.

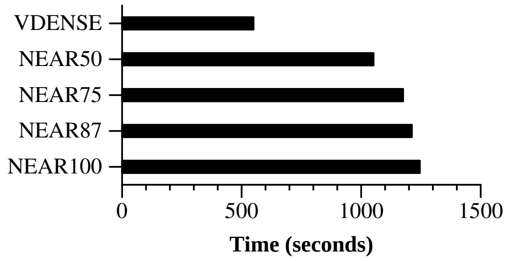

Figure 11: Uniform traversal (varying fill factor): Elapsed time.

6.3 Experiment 3: Working Set Traversals

Our second traversal method is called the Working Set traversal because it models an application that is active in different portions (working sets) of the database at different times. The Working Set traversal repeatedly picks a file to be its current working set, visits some number of clusters from that file only, and then chooses another file (with replacement). As before, the traversal visits 25,600 clusters in all, visiting 1,600 clusters in each file before moving on to another file. To examine the effects of memory size and disk-space utilization on Working Set traversals, we ran the traversals with buffer pool sizes varying from 200 pages (1.6 Mbytes) to 1,152 pages (9 Mbytes).

Figure 12 shows the elapsed-time results for the Working Set traversals. The most striking aspect of this graph is the range of performance observed for NEAR87 and NONE. With a small buffer pool, they perform up to three times slower than VMIN, VDENSE and FIXED, but with a large buffer pool, they are actually the fastest algorithms. As in the previous experiments, the small buffer pool results are explained by the fact that clusters end up being split over many pages with NEAR87 and NONE. As a result, if the individual working sets do not fit in the buffer pool, these methods have a very high I/O cost per cluster read. The greatly improved performance for NEAR87 and NONE with the large buffer pool is explained by their better space utilization; if the working sets do fit in the buffer pool, then the only remaining I/O cost is the cost of switching from one working set to another. This cost is lowest when space utilization is the highest, as higher space utilization implies that each working set covers fewer pages, thereby reducing the number of I/O operations needed to transfer the working set from disk to memory. Since NEAR87 and NONE yield the best space utilization, they perform slightly better than VMIN and VDENSE at this end of the buffer pool spectrum.

The difference in performance between FIXED and the Vcluster algorithms is dwarfed by the NEAR87 and NONE curves in Figure 12, but it is significant nonetheless. The right-hand side of Figure 12 focuses on the portion of the graph for larger buffer pools. FIXED is up to 38% slower than VMIN and VDENSE. The worse performance of FIXED is due to its worse space utilization, as more page I/Os are required to read each working set. In other words, FIXED not only suffers from poor disk-space utilization, but it also exhibits inferior run-time performance due to poor utilization of disk/memory bandwidth.

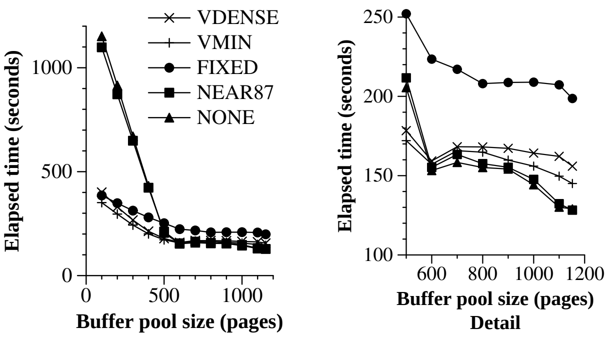

Figure 12: Working Set traversal: Elapsed time.

6.4 Experiment 4: Updates

Our fourth experiment has two parts. We first measure the performance of a workload that creates and deletes clusters and objects, and then we measure the extent to which the algorithms preserve clustering by rerunning Experiment 2 on the updated database. The Update workload is designed to hold the size of the database approximately constant. Each tree operation is chosen with equal probability to be *create*, *delete*, *grow*, or *shrink*. The number of leaves added or removed by *grow* and *shrink* operations was chosen uniformly from the range of 4–8.

The results of running this workload for 25,600 operations are shown in Figure 13. As expected, VMIN and VDENSE do have a somewhat higher cost than the other methods; VMIN is roughly 27% slower than FIXED. Interestingly, NONE is also slower than FIXED or NEAR87. This effect can be explained by the fact that the *destroy* and *shrink* operations are similar to traversals in that they visit the root of a tree and then visit one or more of its leaves. Better clustering makes these operations faster.

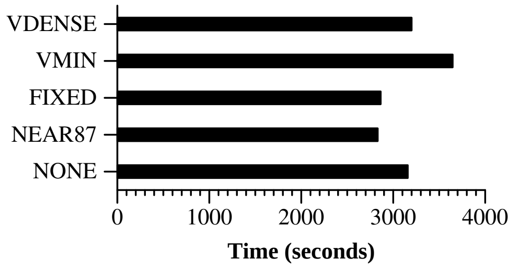

Figure 13: Update workload: Elapsed time.

Figures 14 and 15 show the effect of updates on clustering by comparing the space utilization and elapsed time for a Uniform traversal (Experiment 2) before and after the Update workload was applied to the database. NEAR87 shows the smallest degradation of performance (9%), partly because the free-space created on each page by deletions helps near hints to succeed, but its post-update traversal performance is still worse than any other method except NONE. In contrast, the performance of VMIN degrades by just 10%, and in fact it manages to give the best traversal performance of all.

FIXED suffers the greatest performance degradation (37%) primarily due to cluster overflows. Over time, more and more trees become spread over multiple clusters (pages).

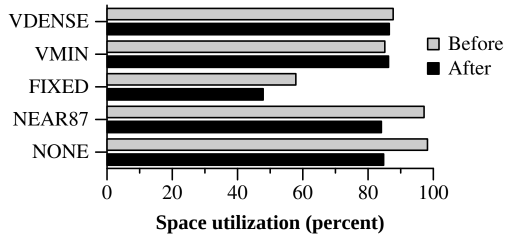

Figure 14: Update workload: Space utilization before and after updates.

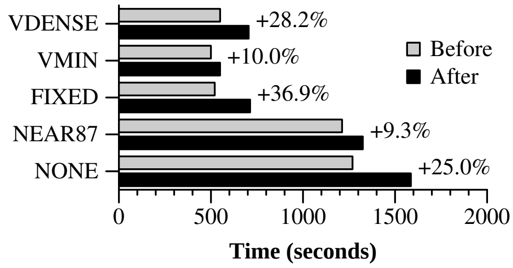

Figure 15: Update workload: Elapsed time before and after updates.

Shrink operations are unlikely to remove clusters since a cluster is removed only if it becomes empty. If a tree spans multiple clusters, one of which is nearly empty, a leaf chosen randomly to be deleted from the tree is unlikely to lie in the nearly empty cluster. Therefore, *shrink* operations tend to decrease utilization, causing a 10% drop in space utilization (from 58% to 48%) for FIXED.

The 28% drop in performance for VDENSE is due to its willingness to split clusters to maintain space utilization. Apparently, the attempt to rejoin clusters during reorganization is not particularly successful. VDENSE and VMIN do manage to hold space utilization nearly constant. Utilization for NEAR87 and NONE drop to near the HY(n,u) parameter u.

6.5 Experiment 5: Size Hints

Experiment 5 is designed to measure the extent to which our algorithms can take advantage of size hints. The Create and Update workloads were modified to supply size hints on *create* and *grow* operations. Since any mechanism that accepts hints must allow for the possibility that the hints are wrong, we measured the performance with both valid and invalid hints. The methods used for calculating size hints are summarized in Table 3. For *create*, the CLOSE, HALF, and DOUBLE size hints are computed from the number of leaves in the cluster (which is known when the cluster's root is created) by assuming an average leaf-node size of 500 bytes and a root-node size of 428 bytes. CLOSE is the actual estimate, while HALF and DOUBLE are each deliberately wrong by a factor of two. AVG (average) always supplies a size hint of 5,428 bytes, the overall average tree size. Hints for *grow* operations are computed similarly, based on the number of leaves to be added.

Figure 16 shows the effect of size hints on Create performance for VMIN. It also includes data for FIXED, NEAR87, and NONE from Figure 7 for comparison. The VMIN al-

Size hint calculation	*Label*
Size hint = Estimated tree size	CLOSE
Size hint = Estimated tree size / 2	HALF
Size hint = Estimated tree size $*$ 2	DOUBLE
Size hint = Overall average tree size	AVG
No size hints	NOHINT

Table 3: Methods for calculating size hints and their labels.

gorithm benefits significantly from accurate size hints, while successfully tolerating incorrect hints. The CLOSE hints decreased the elapsed time by 28%, bringing it to within 20% of the time to create the database with fixed-size clusters. Figure 17 shows that size hints had a similar but less pronounced effect on the Update workload. We also measured the performance of Uniform traversals on databases created with size hints (not shown). We saw practically no variation, showing that size hints have little impact on the quality of the resulting database layout. Results for VDENSE (also not shown) were similar.

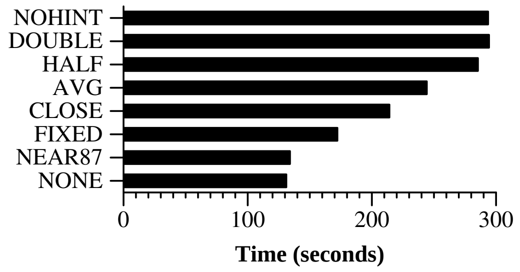

Figure 16: Create workload with size hints: Elapsed time (VMIN).

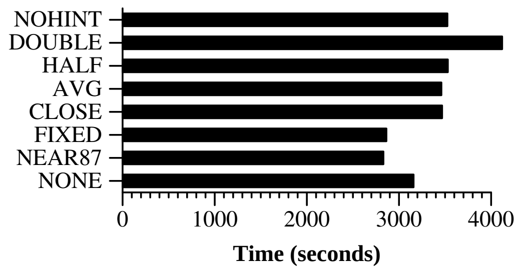

Figure 17: Update workload with size hints: Elapsed time (VMIN).

7 Conclusions

Effective fine-grained clustering algorithms for object-oriented database systems must not just avoid splitting clusters across multiple pages—they must also provide good space utilization and must be easy to use. Two general-purpose clustering mechanisms available in current commercial and experimental object-oriented database systems are near hints and fixed-size clusters. Near hints are found in virtually every OODB system available today, as they are easy to implement, easy to use, and achieve excellent space utilization. Unfortunately, as we have shown, they don't work at at all well for clustering. They are only marginally better than nothing when populating a database, and they are no help for maintaining clusters over time. Fixed-size clusters achieved excellent clustering in some of our tests, but in general they are cumbersome to use and can lead to poor space utilization due to internal fragmentation.

To address the shortcomings of near hints and fixed-size clusters, we introduced Vclusters, a new fine-grained, general-purpose clustering mechanism that supports clusters whose sizes can vary dynamically according to the amount of data that they contain. Vclusters are designed to be as easy to use as near hints, but to be much more effective than near hints at clustering data and supporting incremental cluster maintenance. They can be used either as the back-end of an automatic clustering algorithm or directly by application programmers. We presented two specific algorithms for implementing Vclusters. Both algorithms attempt to avoid splitting clusters while maintaining good space utilization. The algorithms differ only when these goals conflict; each algorithm favors one goal at the possible expense of the other.

We implemented both algorithms in Shore and compared them to near hints and fixed-size clusters. The results of our performance study point out the ineffectiveness of near hints, even when used in conjunction with a fill factor. Traversals of databases built using our Vcluster algorithms were more than twice as fast as traversals of databases built using near hints, except in simple cases where the database was built with no interleaving. Traversal performance for near hints improved by about 18% when a fill factor of 50% was used, but Vclusters still outperformed near hints by nearly a factor of two. Our results also show that for some workloads, such as our Uniform traversals, it is possible for fixed-size clusters to be as effective as Vclusters in terms of traversal performance. However, fixed-size clusters were shown to give very poor space utilization in all of our experiments. As a result, other traversals, such as our Working Set traversals, were up to twice as slow with fixed-size clusters as they were for Vclusters.

Vclusters gave superior performance on all of the read-only traversals in our performance study. Although this performance came at the expense of somewhat poorer performance for update-intensive workloads, our results suggest that Vclusters will provide better overall performance than near hints or fixed-size clusters when traversals are more common than updates. A comparison of our two Vcluster algorithms shows that Vcl-minsplit has a higher update cost than Vcl-dense but it is also more effective at improving traversal performance, even for heavily updated databases.

Finally, we saw that the extra cost of creating Vclusters can be significantly decreased with the provision of accurate size hints when clusters are created. For example, accurate size hints improved creation performance as much as 38% for Vcl-minsplit and also helped with updates (although not as much). On the other hand, inaccurate size hints were shown to be harmless, both in terms of performance and in terms of the quality of clustering produced in the face of inaccurate hints.

Several fertile areas remain for future work. One such area is related to the concurrency and recovery aspects of moving objects. While we mentioned possible approaches here, experimentation is needed to explore their tradeoffs. More generally, experimentation with true multiuser workloads is needed to investigate the impact of locking, particularly for create-intensive workloads, to verify that no unexpected concurrency bottlenecks are introduced by our clustering algorithms.

References

[BD89] V. Benzaken and C. Delobel. Dynamic Clustering Strategies in the O_2 Object-Oriented Database System. Technical Report TR 34-89, Altaïr, August 1989.

[BKKG88] J. Banerjee, W. Kim, S. Kim, and J. Garza. Clustering a DAG for CAD Databases. *IEEE TSWE*, 14(11):1684–1699, November 1988.

[BSI94] E. Bertino, A. Saad, and M.A. Ismail. Clustering Techniques in Object Bases: A Survey. *Data and Knowledge Engineering*, 12(3):255–275, May 1994.

[CDK+94] M. Carey, D. DeWitt, C. Kant, , and J. Naughton. A Status Report on the OO7 OODBMS Benchmarking Effort. In *Proceedings International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 414–426, 1994.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proceedings ACM SIGMOD*, pages 12–21, Washington, DC, May 1993.

[CH91] J. Cheng and A.R. Hurson. Effective Clustering of Complex Data in Object-Oriented Database Systems. In *Proceedings ACM SIGMOD*, pages 22–31, 1991.

[CK89] E. Chang and R. Katz. Exploiting Inheritance and Structure Semantics for Effective Clustering in an Object-Oriented Database System. In *Proceedings ACM SIGMOD*, pages 348–357, 1989.

[CSL+90] M. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson. An Incremental Join Attachment for Starburst. In *Proceedings VLDB*, pages 662–673, 1990.

[Deu90] O. Deux. The Story of O_2. *IEEE TKDE*, 2(1):91–108, 1990.

[DKH90] P. Drew, R. King, and S. Hudson. The Performance and Utility of the Cactis Implementation Algorithms. In *Proceedings VLDB*, pages 135–147, 1990.

[GA94] O. Gruber and L. Amsaleg. Object Grouping in Eos. In M. Öszu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 117–131. Morgan Kaufmann, 1994.

[GKKM94] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-Based Clustering in Object Bases: From Theory to Practice. In *Proceedings FODO*, pages 301–316, 1994.

[HCL+90] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE TKDE*, 2(1):143–160, March 1990.

[HK89] S. Hudson and R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object Oriented DBMS. *ACM TODS*, 14(3):291–321, September 1989.

[HZ87] M. Hornick and S. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM TOIS*, 5(1):273–285, 1987.

[McA97] M. McAuliffe. *Storage Management Methods for Object Database Systems.* Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[MCS96] M. McAuliffe, M. Carey, and M. Solomon. Towards Effective and Efficient Free Space Management. In *Proceedings ACM SIGMOD*, pages 389–400, Montréal, Canada, June 1996.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Grain Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, March 1992.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK94] W. McIver, Jr. and R. King. Self-Adaptive On-Line Reclustering of Complex Object Data. In *Proceedings ACM SIGMOD*, pages 407–418, 1994.

[Obj95] Object Design, Inc., Burlington, MA. *ObjectStore C++ API User Guide, Release 4.0*, June 1995.

[Sal86] B. Salzburg. *An Introduction to Data Base Design.* Harcourt Brace Jovanovich, 1986.

[Söd81] L. Söderlund. Concurrent Database Reorganization—Assessment of a Powerful Technique Through Moduling. In *Proceedings VLDB*, pages 499–509, 1981.

[SS90] K. Shannon and R. Snodgrass. Semantic Clustering. In *Proceedings of the Fourth International Workshop in Persistent Object Systems. Martha's Vinyard, MA*, pages 361–374, September 1990.

[TN91] M. Tsangaris and J. Naughton. A Stochastic Approach for Clustering in Object Bases. In *Proceedings ACM SIGMOD*, pages 12–21, 1991.

[TN92] M. Tsangaris and J. Naughton. On the Performance of Object Clustering Techniques. In *Proceedings ACM SIGMOD*, pages 144–153, 1992.

[ZSL97] C. Zou, B. Salzburg, and R. Ladin. Back to the Future: Dynamic Hierarchical Clustering. In *Proceedings ICDE*, 1997.