A Highly Effective Partition Selection Policy for Object Database Garbage Collection

Jonathan E. Cook, *Member*, *IEEE Computer Society*,
Alexander L. Wolf, *Member*, *IEEE Computer Society*, and
Benjamin G. Zorn, *Member*, *IEEE Computer Society*

Abstract—We investigate methods to improve the performance of algorithms for automatic storage reclamation of object databases. These algorithms are based on a technique called *partitioned garbage collection*, in which a subset of the entire database is collected independently of the rest. We evaluate how different application, database system, and garbage collection implementation parameters affect the performance of garbage collection in object database systems. We focus specifically on investigating the policy that is used to select which partition in the database should be collected. Three of the policies that we investigate are based on the intuition that the values of overwritten pointers provide good hints about where to find garbage. A fourth policy investigated chooses the partition with the greatest presence in the I/O buffer. Using simulations based on a synthetic database, we show that one of our policies requires less I/O to collect more garbage than any existing implementable policy. Furthermore, that policy performs close to a locally optimal policy over a wide range of simulation parameters, including database size, collection rate, and database connectivity. We also show what impact these simulation parameters have on application performance and investigate the expected costs and benefits of garbage collection in such systems.

Index Terms—Garbage collection, storage reclamation, storage management, object databases, partition selection, performance evaluation.

1 Introduction

Object database management systems (ODBMSs) can be viewed as an integration of research results from the areas of database management systems and programming language systems. The goal of the integration is to support the definition of a richer set of types for persistent data and to support the manipulation of those data using a more powerful, programming-language-like model of computation [19]. Two concepts that were extensively developed in the programming language area and that are now profitably employed in ODBMSs are direct support for complex, highly interconnected data and a notion of object identity separate from object value.

The introduction of these two concepts, however, has greatly complicated a critical performance aspect of database management systems, namely the reclamation of storage for persistent, secondary-memory objects that are no longer accessible. As shown by decades of experience with network and relational databases, the presence of inaccessible data, while not affecting the functional behavior of an application, can have an impact on its performance, since such data increase the effective size of the database and can increase access time. As a result, most database systems provide a "compact" or "reorg" operation to allow database administrators to reduce the database storage usage and improve locality of reference.

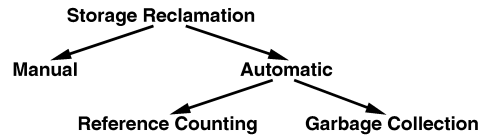


Fig. 1. Approaches to storage reclamation.

Designers of ODBMSs actually have some choice in how storage is reclaimed (see Fig. 1). The two most basic options are *manual reclamation*, in which explicit deallocation commands are issued by an application, and *automatic reclamation*, in which the underlying management system itself directs a process of "discovering" implicitly inaccessible objects.[1] Within automatic reclamation there is a choice between *reference counting*, in which the discovery of inaccessible objects can be made using only local information (i.e., the reference count), and *garbage collection*, in which discovery is made using global information that is obtained (at least conceptually) by traversing the graph of "live" objects.

Each approach has its advantages and its disadvantages. Manual techniques give the application fine control over the timing and extent of reclamation, but run the risk of insidious errors, such as dangling references and unreclaimable space. Reference counting techniques are attractive because of their basis in local information, but may be ineffective, without complex enhancements, in the presence of cyclic data.[2]

- *J.E. Cook is with the Department of Computer Science, New Mexico State University, Las Cruces, NM 88003. E-mail: jcook@cs.nmsu.edu.*
- *A.L. Wolf and B.G. Zorn are with the Department of Computer Science, University of Colorado, Boulder, CO 80309. E-mail: {alw, zorn}@cs.colorado.edu.*

Manuscript received 25 Aug. 1994; revised 1 Nov. 1995.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104440.

1. For comprehensive surveys of the field of storage reclamation, see the papers by Cohen [12] and Wilson [32].

2. Because the application programming model of object databases follows object-oriented programming practices closely, we anticipate that the same problems that arise in object-oriented programming, which include cyclic data structures, are also likely to arise in object database programs.

Garbage collection techniques are appropriate and safe for all kinds of data, but their basis in global information makes them difficult to efficiently implement in a distributed setting. We note that a recently proposed standard for ODBMSs suggests using manual reclamation for some of the programmatic interfaces to an ODBMS, while at the same time suggesting garbage collection for other interfaces [10].

Although the choices described above may seem few, they represent only the surface of the design space available to ODBMS developers. And while a similar problem has been studied for three decades by designers of programming language systems in the realm of transient, primary-memory (heap) objects, only recently has a strong interest developed in formulating the storage reclamation techniques that are commensurate with the size, complexity, and stability characteristics of persistent, secondary-memory objects in ODBMSs [1], [4], [5], [20], [21], [33]. In fact, it has been shown that the reclamation algorithms developed for programming language systems are, as currently formulated, inappropriate for use on object databases [5], [33]. These results are not surprising, given the large number of differences—including transactions, sharing, persistence, and volume of data—that exist between programming language and database systems.

This paper reports on the design and evaluation of several new policies related to storage reclamation that are specifically intended for use by ODBMSs. Our designs apply to a class of algorithms known as *partitioned garbage collection* algorithms [33], in which a subset of the entire database is collected incrementally and independently of the rest. The policy we investigate is the one that selects a partition (i.e., the subset) of the database to examine during a particular activation of the garbage collector. We evaluated the policies using simulations based on synthetic database applications that create, access, and modify objects. Our results show that the partition selection policy can significantly affect application performance. Furthermore, we show one of our policies has performance close to a locally optimal, but impractical-to-implement, policy over a wide range of ODBMS, collector, and application parameters.

In the remainder of this section, we provide some important background to the work described in this paper. In particular, we discuss some lessons learned from programming-language-system algorithms and clarify the notions of partitioning and partitioned garbage collection.

1.1 Background

Useful insights into storage reclamation for ODBMSs can be gained from the programming-language-system experience. One fundamental insight is that the performance of garbage collection over a large address space is improved if the objects in the address space are partitioned into groups, where storage in each group can be reclaimed independently [3], [22]. Thus, only a subset of a potentially huge set of objects needs to be considered at any point by a collector. There are two key advantages to collecting a subset of the total object space: the locality of reference of the collection algorithm is greatly improved, substantially reducing paging I/O operations, and the disruption caused by interfering with the application during garbage collection is reduced. A corollary of this reasoning is that the collection algorithm can be made incremental with respect to the application; reclamation of the entire database amounts to a series of individual collections of portions of the database.

One obvious and important question that arises is: What criterion should be used to partition the object space? In general, the answer is a criterion that will make each collection maximally effective—that is, one that gathers together objects likely to contemporaneously become garbage. In the programming language domain, garbage collection algorithms are dominated by those that use object age as the criterion, since empirical data on programs clearly demonstrate that objects of similar age usually exhibit similar lifetimes [27], [34]. Thus, these algorithms are referred to as *generational* collection algorithms [22], [30]. In ODBMSs, no such universal criterion has yet emerged. In fact, a number of different criteria for partitioning an object database—predating the recent interest in garbage collection and hence not designed with garbage collection in mind—have already been built into those systems. Three such criteria are:

1) partition objects based on access patterns (e.g., [25], [29]);
2) partition objects based on the unit of transfer between a server and a client (e.g., [16]); and
3) partition objects to increase locking granularity and thereby decrease overhead (e.g., [8]).

The previous discussion brings to light an important distinction between programming language systems and ODBMSs that makes the design of garbage collection algorithms for ODBMSs especially challenging. In programming language systems, designers of collection algorithms have complete control over storage management. For example, they are free to decide how to partition the object space. They are also free to decide where objects should be relocated in primary memory as part of the actual recovery of space. In contrast, designers of garbage collection algorithms for ODBMSs are constrained in their choices, since, as pointed out above, those choices are historically driven by performance considerations unrelated to storage reclamation. Thus, in ODBMSs, any new garbage collection algorithm must be carefully designed to integrate with traditional database storage organization and performance concerns.

While the criterion used to partition the object space is essentially a given, there are other important policies to consider in designing a garbage collection algorithm. These policies include partition selection, collection rate, how to reclaim space (e.g., copying versus noncopying collection), and how to deal with distributed, cyclic garbage. In this paper, we focus on *partition selection*, the selection of which partition to examine for garbage during a particular collection. Since existing database partitioning schemes are not

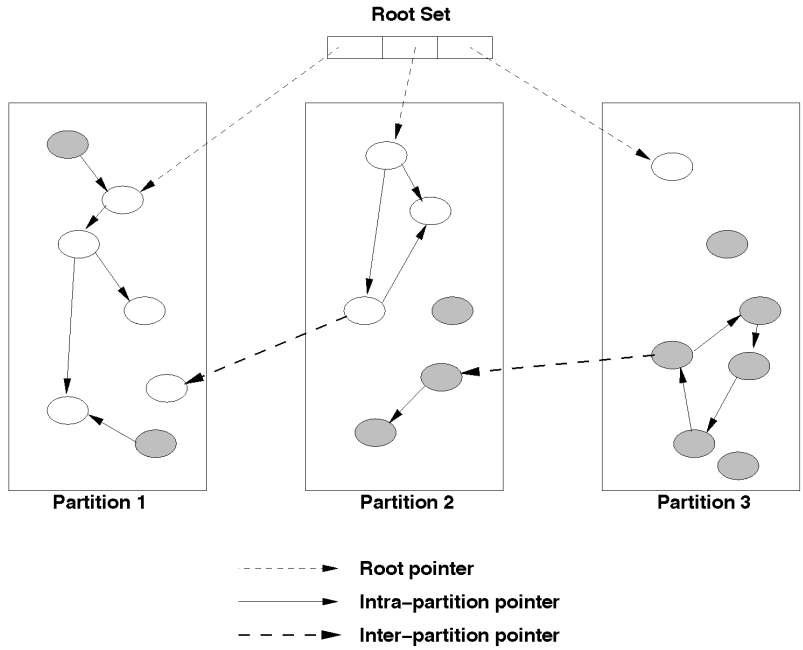


Fig. 2. Example of a partitioned object database.

intended to support the efficient collection of garbage, we must invent new ways to effectively find partitions likely to contain garbage. Other issues, such as the best way to reclaim distributed cyclic garbage that spans partitions, are not addressed by our current work.

We illustrate the problem in Fig. 2. In the figure, we depict an object database that has been divided into three partitions for some reason (e.g., to improve access performance). The figure shows how objects in the database are interconnected (indicated by the arrows between objects), where some of the connections are within a partition (solid arrows) and others are between partitions (dashed arrows). We refer to such connections simply as *pointers*. The objects that are still in use (white objects) are reachable from a so-called *root set* by a transitive traversal of the pointers. The root set serves as the entrée into the database for applications. The remaining objects, which are not reachable from the root set, are garbage (shaded objects) waiting to be collected.

At each (incremental) collector activation, a partitioned garbage collection algorithm uses a selection policy to choose which partition(s) in the database to examine for inaccessible objects currently in that partition. For example, if Partition 1 in the figure were selected for collection, then two objects would be reclaimed. To correctly collect one partition without entirely scanning the others, information must be kept about object pointers that cross partition boundaries (i.e., interpartition pointers). For example, if the pointer from Partition 2 into Partition 1 was not considered when Partition 1 was collected, a live object would be incorrectly considered garbage. Also note that the shaded (dead) object in Partition 2 pointed to from Partition 3 can only be reclaimed after Partition 3 has been collected and the interpartition pointer from Partition 3 to Partition 2 has been eliminated. The space and time overhead of maintaining sets of interpartition pointers is one cost of a partitioned collection approach. This example also illustrates the importance of the partition selection policy. Intuitively, selecting Partition 3 for collection is likely to result in the best performance since more inaccessible objects will be reclaimed.

The evaluation described in this paper compares the performance of four quite different partition selection policies, including three new ones that we devised and one previously existing one that we enhanced. Three are based on the intuition that the values of overwritten pointers provide good hints about where to find garbage. This intuition has not yet been explored, even within the programming language community. The fourth policy is based on the intuition that collecting the partition with the greatest presence in the I/O buffer will result in fewer I/O operations during garbage collection.

The remainder of this paper has the following organization. In Section 2, we discuss related work that investigates ODBMS garbage collection. We then describe the policies for partition selection in Section 3 and the methods used to evaluate the algorithms derived from the policies in Section 4. The test database is described in Section 5 and, in Section 6, we present the results of the evaluation. We summarize our conclusions and discuss future work in Section 7.

2 Related Work

Most research investigating the use of garbage collection in ODBMSs has not been explicitly directed at partitioned garbage collection algorithms. Although some of the proposed algorithms do act incrementally on subsets of the database [4], [6], [21], [23], the partitioning aspects of those algorithms have been treated secondarily at best.

Techniques for garbage collection may be classified by whether the algorithm relocates objects during execution.

Copying algorithms are allowed to relocate objects as needed, whereas noncopying algorithms leave objects in place. Campin and Atkinson describe a breadth-first, copying garbage collector used in the PS-Algol persistent programming language [6]. They chose to use a copying collector to simplify error recovery when a crash occurs during collection. Their collector performs garbage collection across multiple databases, allowing the collection of only one database at a time, if desired.

Matthews describes a mark-and-sweep garbage collector for a persistent version of ML called Poly [23]. The collector performs a complete reachability analysis, but at the granularity of a page. The analysis is neither depth-first nor breadth-first. Rather, it can be said to be page-first, since all intrapage object references are followed before any interpage object references are considered. The advantage of this scheme is that it minimizes the number of times a page needs to be read from secondary memory.

Kolodner, Liskov, and Weihl propose using copying garbage collection in a "stable heap" [20]. Their collection algorithm, atomic garbage collection, guarantees heap consistency during a system crash and interacts with the recovery system to insure correctness. More recently, Kolodner and Weihl propose an enhancement of their original algorithm that is also incremental and works on stock hardware [21]. As with the original algorithm, their collector accommodates system failures.

Björnerstedt describes an algorithm for collecting a distributed persistent object store [4]. His algorithm decouples collection of objects in primary memory from those in secondary memory, collecting secondary-memory objects using a distributed mark-and-sweep approach.

Butler investigates the performance of different persistent storage management algorithms using probabilistic models of program reference and update behavior [5]. For a number of dynamic storage allocation algorithms, she shows the expected I/O costs based on complex formulas. Her results show that the incremental collection proposed by Baker [2] provides significant advantages over other traditional primary-memory collection algorithms when applied to object databases. Butler did not consider partitioned collection algorithms.

A somewhat different thread of related research has occurred in the area of garbage collection for programming languages that support persistence and transactions. Detlefs investigates the use of concurrent, atomic garbage collection for transaction-based programming languages [15]. Nettles and O'Toole have developed a concurrent, replicating garbage collection algorithm to support persistence in ML programs [24].

In distributed object databases, reclaiming distributed cyclic garbage will be a significant concern. While our current work does not directly address this issue, there is a large body of previous work that has explored reclaiming distributed cyclic garbage (e.g., see [17], [26]). Our work can be extended with these techniques.

Partition-based collection algorithms are an extension of generational algorithms used in primary-memory garbage collection. In a 1977 dissertation, Bishop investigated the possibility of garbage collection in a very large address space [3]. Bishop suggested dividing a large address space into areas and maintaining lists of interarea references to allow each area to be collected independently. Lieberman and Hewitt applied Bishop's approach to conventional language systems such as Lisp [22]. They enhanced Bishop's idea with the suggestion that objects should be segregated by age, inventing generational garbage collection.

Amsaleg, Franklin, and Gruber consider an approach to partitioned garbage collection based on a noncopying mark-and-sweep algorithm [1]. In their approach, which operates in a client-server environment, garbage collection is incremental and runs concurrently with client transactions. Furthermore, they avoid callbacks and reduce the logging necessary to make the algorithm recoverable. In their work, they do not investigate partition selection policies.

Yong, Naughton, and Yu present a comprehensive evaluation of incremental, reference counting, and partitioned garbage collection algorithms in client/server persistent object stores [33]. They conclude that an incremental partitioned collection algorithm shows the best performance based on a number of metrics including scalability and locality improvement.

Our work extends and complements the work of Yong, Naughton, and Yu. We agree with their general conclusion that an incremental, partitioned algorithm is the most appropriate way to collect an object database. We take the next step in this work by investigating the specific problem of partition selection.

3 Garbage Collection Policies

Because garbage collection of object databases is an area of relatively recent interest, there is still much to understand about the design of appropriate algorithms. To make the investigation of the space of design parameters more systematic, we have identified independent and separable policy decisions (of which partition selection is one) that must be made when designing and implementing an ODBMS garbage collector.

Many of these policy decisions can be made by the ODBMS implementor; a handful of the decisions should be left to the database administrator as tuning parameters. Every one of the policies can significantly affect the performance of the algorithm. Table 1 summarizes some of the policies contributing to a partitioned garbage collection algorithm.

As mentioned above, our concern in this paper is primarily with one of the policies, namely the selection of a partition to collect (i.e., the last entry in Table 1). Our approach to studying this policy is to make reasonable decisions for the other policies and to hold them constant while varying the selection policy. We describe our decisions in Section 4.

For the purposes of this paper, most of the other policies should be understandable from their brief description in Table 1. The only one that deserves further explanation is the policy concerned with maintaining the interpartition pointers. These pointers are commonly stored in a data structure called the *remembered set* [30], which records all interpartition pointers into a partition on a per-partition

TABLE 1
PARTIAL LIST OF POLICIES CONTRIBUTING TO AN ODBMS GARBAGE COLLECTION ALGORITHM

Policy	Some Alternatives
How to reclaim space	• Copying • Non-copying
How database partitions relate to GC partitions	• They are the same • DB partition contains > 1 GC partition • GC partition contains > 1 DB partition
How to traverse objects during collection	• Breadth-first • Depth-first • Partition-first • Page-first
When to perform collection	• When more space is needed • When performance degrades • When garbage is created • Opportunistically
When to grow database	• When free space is exhausted • When collector efficiency is low
How collector interacts with applications	• Locks entire database • Executes concurrently with applications
How to record inter-partition pointer information	• Sequential store buffer • Page or card marking
Which partition to select for collection	• INBUFFER • MUTATEDPARTITION • UPDATEDPOINTER • RANDOM • WEIGHTEDPOINTER

basis. Each time a write occurs there is a possibility that an interpartition reference has been created or destroyed. The remembered set is, therefore, maintained by identifying when interpartition references are created or destroyed, a process occurring at the so-called *write barrier*. Language system techniques for maintaining the remembered sets and the write barriers exist and can be applied in the domain of object databases. A number of well-known and effective implementations, including those mentioned in the table, have been evaluated and compared [18]. In addition, most object databases already make use of the write barrier for purposes such as concurrency control and recovery. Furthermore, since secondary-memory writes involve a great deal more overhead than primary-memory writes, the additional CPU impact of maintaining the write barrier in an ODBMS is less significant than it is for a programming language system, where the overhead has been already found acceptable. In any event, all algorithms based on partition garbage collection must maintain remembered sets and so their implementation will not differ among the policies we examine.

3.1 Partition Selection Policies

If we assume the goal of garbage collection is to reclaim the most garbage, then the ideal partition selection policy would select the partition that contains the most garbage. Because it is unrealistic for an implementation to actually have this information, partition selection policies use heuristics, attempting to guess at the optimal partition. We investigate the relative performance of three policies based on three different heuristics for finding the partition with the most garbage, MUTATEDPARTITION, UPDATEDPOINTER, WEIGHTEDPOINTER. We compare these with a fourth policy, INBUFFER, that chooses, not the partition with the most garbage, but rather the partition with the greatest presence in the I/O buffer, under the assumption that the savings in I/O operations will outweigh the costs of choosing a suboptimal partition to collect. We compare the four policies with three other policies, RAMDOM, ORACLE, and NOCOLLECTION, that are intended to place upper and lower bounds on the effect of partition selection on application performance. All seven selection policies are described below.

MUTATEDPARTITION. Under this policy, the partition selected is the one in which the most pointers have been updated since the last collection. The rationale for this policy is that the partition in which the most pointer mutation has occurred is also likely to contain the most garbage, making collection more efficient. This is an enhancement of the Yong, Naughton, and Yu policy, which selects the partition that had been mutated the most, without regard to whether the mutations were to the partition's pointers or to its data. Our version should lead to better performance, since pure data mutations, which do not affect object connectivity and, hence, cannot create garbage, are not considered.

To implement this policy, we perform the following operations. When a write occurs, we determine if the value being written is a pointer, and if it is, we increment the counter associated with the partition being written into. When a collection is required, we select the partition with the most mutations. After the collection of a partition, we zero the counter for the partition collected and continue.

This policy is inexpensive to implement for two reasons. First, it does not require any additional storage except for a single counter per partition. Second, it does not add to the number of I/O operations performed by the collector, because a partitioned collector must maintain the write barrier in any event. The per-partition mutation count is a small array that can easily be maintained in primary memory.

UPDATEDPOINTER. This policy is based on the observation that when a pointer is overwritten, the object it pointed to is more likely to become garbage. Reference counting algorithms carefully record such pointer deletions; with this policy, pointer deletions are used as a hint to determine a partition that contains garbage. In particular, this policy records, for each partition, the number of overwritten pointers that pointed into that partition since the last garbage collection. The partition with the most overwritten pointers into it is selected for collection.

Another important aspect of this policy is that it incorporates information gathered during the collection of a partition. In particular, when an object in a partition is reclaimed, its pointers are conceptually overwritten, as they are no longer needed. If such pointers point to other partitions, the overwritten

pointer count for these partitions is increased accordingly. In practice, our algorithm maintains an out-of-partition set which indicates which objects in a partition contain pointers to other partitions. As a result, only the objects in the out-of-partition set need to be scanned.

Note that unlike traditional reference counting, UPDATEDPOINTER does not attempt to maintain perfect information about references to objects.

This policy is very similar in cost to MUTATED-PARTITION. In particular, the only difference is that when a pointer write is about to occur, the value of the pointer being overwritten must be read and its partition determined. Once the partition is determined, the count for that partition is incremented.

WEIGHTEDPOINTER. This policy is a refinement of UPDATED-POINTER based on the observation that not all pointers are equal. In particular, when the object connectivity is low (i.e., most objects are pointed to by one other object, as in a tree), loss of connectivity to objects near the root results in many other objects becoming garbage at the same time. Conversely, when pointers to objects far from the roots (and near the leaves) are overwritten, few other objects are likely to become garbage at that time. This partition selection policy augments all pointers with information about their distance from the database root objects; pointers close to the root have a greater weight than pointers far from it.

Our simulated implementation maintains 4 bits of weight information per object—an object's weight is one plus the minimum of the weights of the edges pointing to it. The weight approximates the distance of the object from the root (with a maximum of 16). Like the UPDATEDPOINTER policy, when pointers are overwritten, the partition they point into is noted. In this case, however, an exponentially-weighted sum of pointers into each partition is maintained, where the exponential weight is 2^{16-w}, and the partition with the greatest weighted sum is chosen to be collected. Fig. 3 illustrates the weights for a simple graph of objects. As an example of how the weights are used, if the pointer from A to B in the figure were overwritten, the sum for the partition that B is in would be increased by $2^{16-2} = 16{,}384$.

This policy is somewhat more expensive than UPDATEDPOINTER for the following reasons. First, it maintains 4 bits of additional information (i.e., the weight) per object. Assuming objects average 100 bytes, the space overhead of the weight is 0.5 percent. Second, to maintain the weights correctly, the following must occur at the time a pointer is stored: The from-object's and to-object's weights must be read and then the to-object's weight must be compared to the from-object's weight plus one. If it is greater, the to-object's weight must be updated to reflect the new lower-weight edge pointing to it. This operation is performed transitively if it affects the weights of objects it points to. Finally, if this is a pointer overwrite, the old to-object's partition counter must be updated with the exponential weight.

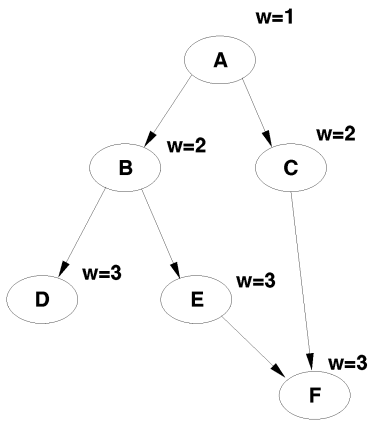


Fig. 3. Object weights in a simple directed graph.

INBUFFER. Unlike the previous algorithms, this algorithm does not attempt to collect the partition that contains the most garbage. Instead, it attempts to directly improve the reference locality of the application by eliminating garbage from pages in the I/O buffer. It does this by counting the number of pages that each partition has in the I/O buffer at the time of collection and selecting the partition with the largest number of pages in the buffer. The intent of this algorithm is to recluster and compact the objects on the pages that are currently being used. Another effect of this algorithm is that fewer pages need to be read in during a garbage collection because some fraction of the pages from the partition are already in the buffer.

RANDOM. This policy selects a partition to collect at random. We include this policy to determine the extent to which clever heuristics improve or degrade the performance of garbage collection.

ORACLE. This policy is not practically implementable, but we include it for purpose of comparison. Using an oracle (provided by our simulation system), this policy always correctly selects the partition that contains the most garbage. Note that at each collection, ORACLE selects the currently optimal partition to collect, but that selection does not necessarily make the policy globally optimal, either in terms of I/O costs or storage utilization, over all collections.

NOCOLLECTION. This policy does not perform any garbage collection. Instead, when more space is needed, additional partitions are allocated. This policy establishes an upper bound on the amount of space that the application uses. We also use this policy to determine the degree to which garbage collection improves the locality of reference by compacting live objects. In terms of space consumption, the other policies typically fall somewhere between NOCOLLECTION and ORACLE.

When combined with decisions for the other policies listed earlier, these seven partition selection policies result in seven different garbage collection algorithms. For purposes of evaluation, the algorithms we investigate select only a

single partition during a particular garbage collection. A full implementation might allow more than one partition to be collected at a time, if doing so was determined to be of importance.

4 Evaluation Method

In this section, we describe the experiments that we conducted to evaluate the different partition selection polices. We first describe our choices for the other collection algorithm policies and then describe the simulation techniques used in comparing the algorithms.

4.1 Complete Partitioned Garbage Collection Algorithm

To fairly compare different partition selection policies, we must make decisions about the other aspects of the algorithm (see Table 1) and hold them constant. Here, we document the decisions made.

Reclaiming Space. The partition selection policies are evaluated in the context of a copying garbage collector. This choice allows garbage collection to not only reclaim the space occupied by garbage but also to compact the collected partition's live objects for improved reference locality. Copying is done in a breadth-first traversal from the partition's roots, iterating over the roots one at a time. Breadth-first copying was chosen to preserve the object placement policy of the test database (see Section 5). Pointers leaving the collected partition are not traversed.

By compacting live objects in a partition during collection, internal fragmentation is eliminated. The storage overhead of these algorithms includes space occupied by unreclaimed garbage and external fragmentation caused by managing storage in units of partitions.

Partition Organization. Following Yong, Naughton, and Yu, we chose to partition objects physically, segmenting the address space into contiguous partitions. Partition size was varied between 24 and 100 8-kilobyte pages per partition, depending on the size of the database used. We chose these sizes because they resulted in test databases requiring between 15 and 25 partitions. Our main concern was to have enough partitions in the database so that the selection policies could differentiate themselves. Partition size (relative to the database size) also affects how often a collection is performed, since with smaller partitions, each collection reclaims a smaller fraction of the database.

How to Traverse Objects. During a collection, objects are traversed in breadth-first order and copied into an empty partition. After all the live objects in the collected partition have been copied, the collected partition then becomes the empty partition for the next collection. Thus, every algorithm measured maintains one empty partition at all times.

When to Perform Collection. We chose to invoke garbage collection based on the number of pointer stores that the application performs. In particular, garbage collection is triggered after a fixed number of pointer overwrites. This number varied from 50 to 400 overwrites, which resulted in approximately 15 to 90 collections per simulation.

We chose the criterion of pointer overwrites for two reasons. First, because many pointer overwrites result in creation of garbage, the number of stores should be closely correlated to the amount of garbage available for collection. Second, this choice means that collector invocation is independent of the partition choice and allows the different partition selection policies to be compared fairly (i.e., every algorithm performs the same number of collections).

When to Grow Database. The issue of when to grow the database is related to the issue of when to collect. The algorithms we evaluated use the following policy: If an allocation occurs and there is insufficient free space anywhere in the database, a new partition is added. There is no limit on the number of partitions that can be created.

How Collector Interacts with Applications. The two primary issues a collection algorithm needs to address in this area are concurrency and recovery. Because we are concerned with the *relative* performance of partition selection polices, we assume simple mechanisms: The algorithm locks the entire database when collection is performed, and logging for recovery is not supported. Clearly, more sophisticated mechanisms must be provided in actual implementations; proposals for such mechanisms are discussed elsewhere [1], [20], [21], [33].

How to Record Interpartition Pointer Information. The algorithms maintain the remembered sets for interpartition pointers explicitly in auxiliary data structures. To understand what happens when an interpartition pointer is created, we use the following terminology (see Fig. 4). Suppose that pointer P, pointing to an object O2 in partition ToPartition, is written into object O1 in partition FromPartition. First, we maintain a map (the remembered set) from each partition to the interpartition pointers into that partition; thus,

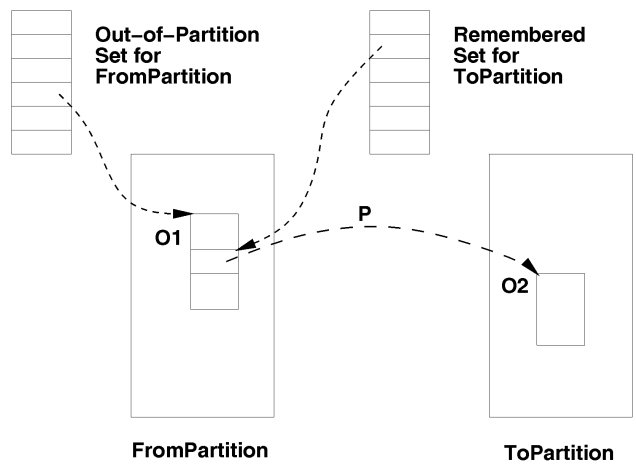


Fig. 4. Data structures used to record interpartition pointer information.

the location of P is added to the remembered set for ToPartition. Second, we maintain a map, for each partition, of all objects containing pointers that point out of the partition; thus, O1 is added to the out-of-partition set of pointers for FromPartition.

The out-of-partition set of objects is maintained for the following reason. When we collect a partition, garbage is reclaimed. Some of that garbage may contain pointers into another partition, in which case the locations of those pointers will be recorded in the remembered sets of the partitions into which the pointers point. When we discover that an element of the out-of-partition set is garbage, we remove the locations of the pointers in it from the remembered set of the partition into which they point. Otherwise, when we collect that partition, we will unnecessarily preserve objects pointed to by garbage.

4.2 Performance Evaluation Method

We use simulations based on a synthetic database to evaluate alternative partition selection policies. This method has been applied effectively in evaluating many kinds of computer systems, including computer architectures [28], programming languages [31], and database systems [11], [33]. To evaluate the performance of storage reclamation in ODBMSs, our simulation system simulates the physical and logical structure of the database implementation being measured. Traces of database application events (e.g., object creations, accesses, modifications) are used to drive the simulations; details appear in [14]. We use synthetic, probabilistic models of application behavior to generate a test database application. Details of the test database are provided in Section 5.

One advantage of using simulation in our evaluation is that we are able to evaluate and compare the performance of impractical to implement, locally optimal algorithms like ORACLE. As a result, we are able to determine how close our heuristics come to the locally optimal solution. Another advantage of using trace-driven simulation is that we are able to investigate the performance of the policies over a broad range of simulation parameters, including the database connectivity, database size, object size, partition size, large-object frequency, and the like. With these results, we have established a clear understanding of the sensitivity of our policies to the values of these parameters. In the future, we plan to also capture traces from existing ODBMS applications and use them to evaluate policy alternatives.

The cost model used to evaluate the performance of the simulated algorithms is based on tracking the number of page I/O operations over the life of the simulation. We simulate the database's I/O buffer and determine the number of disk I/O operations needed for each read and write. To determine when disk I/O operations take place, we simulate a database I/O buffer of a particular size (a parameter to the simulation), using an LRU policy for page replacement and a write-back scheme for updating pages. More detailed cost models can be built that would derive actual disk costs in terms of head seek, rotational delay, and transfer times, or that might model network costs for a distributed or client/server database. It should be noted that garbage collection can improve the clustering of objects, which may improve seek times. Our results, which do not model seek times, will not show this effect. The goal of our simulations is to measure the time-varying behavior of application I/O operations and memory usage and to be able to relate this behavior to the garbage collection policies investigated.

Our simulations all begin with a cold start of the database—that is, starting from an empty database and an empty page buffer. Because we are measuring relative garbage collector performance and not just pure database performance, the only effect that the cold start has on the simulations is to lessen the differentiation among the various algorithms. Cold starts do not qualitatively change the results, since the first few collections will occur when there are not many partitions to choose from, thus making it more likely that a poor policy will pick a good partition. The selection policies all start out showing relatively close performance and it is only once the simulation warms up that the true differentiation among the policies becomes evident. Based on our evaluation of more simulations than are reported here, we did not find that our choice affects the data in a significant manner.

Whenever possible, we evaluate the performance of the policies based on multiple simulation runs that differ only in the initial random number seed. In our results, we present the mean and standard deviation of the values obtained.

5 Test Database Design

Clearly, choices for the design of the test database can affect the results of the simulation. Unfortunately, there is not currently an established or widely accepted body of knowledge about "typical" object databases and applications. While there are indeed several benchmarks emerging, they tend to be weak in modeling database evolution; existing benchmarks are meant to explore aspects of database system performance such as access time, for which evolution is not central. But it is exactly the knowledge about evolution that is required for evaluating garbage collection algorithms. Our simulation system, since it is based on the use of traces, is well positioned to make use of that knowledge once it becomes available. Moreover, with control over characteristics of a test database, such as object size, connectivity, read/write ratio, and the like, we can explore aspects of the problem that may not be possible using only a standard benchmark.

In this section we discuss the structure of the test database, the behavior of the test application, and the characteristics of the database management system we assume. Table 2 summarizes the parameter values used in our simulations.

5.1 Test Database Structure

The test database is a forest of augmented binary trees of objects, where each tree root is itself a root object of the database (see Fig. 5). By augmented binary trees we mean that in addition to the basic tree edges connecting nodes, there are also some number of *dense* edges. The dense edges connect random nodes in the same tree and are created by a random process.

TABLE 2
PARAMETER SETTINGS USED IN THE SIMULATIONS

Test Database Structure		
Parameter Description	Baseline Value	Range of Values
Connectivity (avg. pointers into each object)	1.04	1.02–1.66
Object Size (bytes per object)	100	50–150
Large Object Size (kilobytes per object)	64	not varied
Large Object Fraction (%)	20	not varied
Database Size (megabytes)	5	3–38

Test Application Behavior		
Parameter Description	Baseline Value	Range of Values
Phases Create/Visit/Remove/None Ratio	15:30:50:5	not varied
Tree Traversal Ignore/Depth/Breadth Ratio	30:20:50	not varied
Object Read/Write Ratio	99:1	not varied
Number of Collections	25	15–90

ODBMS Design		
Parameter Description	Baseline Value	Range of Values
Page Size (kilobytes)	8	not varied
Partition Size (pages)	48	24–100
I/O Buffer Size (pages)	48	24–100
Collection Rate (pointer overwrites / collection)	150	50–400

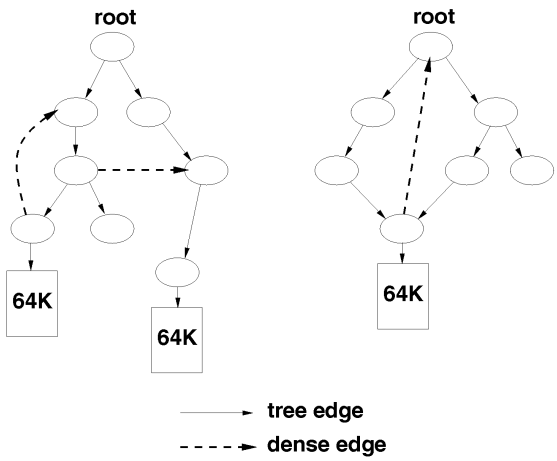


Fig. 5. Illustration of test database structure.

Connectivity. We vary the proportion of dense edges to explore how the connectivity of the database can affect garbage creation and collection. This results in an average of from 1.02 to 1.66 pointers into each object. By comparison, a tree has exactly 1 pointer into each object (except for the root, of course).

With connectivities higher than 1.66, little garbage is created in the database, and the different policies are unable to distinguish themselves given our database structure (as we show in Section 6).

Object Size. Object sizes are randomly distributed around an average of 100 bytes. The distribution is uniform, with bounds at 50 and 150 bytes. We selected this size after looking at previous work and experimenting with a variety of sizes. Butler uses 50-byte objects, although she is expressly modeling Lisp structures [5]. Yong, Naughton, and Yu use objects with an average of about 80 bytes [33]. Cattell's OO1 benchmark averages about 100 bytes per object [9]. Using much larger objects (e.g., on the order of the page size) would tend to reduce the impact of garbage collection on access behavior, since pages would then be more likely to contain either only all garbage or all live objects and, thus, they would only influence database size. We observed this effect in simulation runs that used such larger objects. We do, however, include a few large objects that are 64 kilobytes each. These are always leaf objects and comprise about 20 percent of the space of all objects, in a manner similar to the document nodes in the OO7 benchmark [7].

Database Size. Our simulations vary from databases of 3 to 38 megabytes of allocated objects; garbage collection causes the actual space used by the databases to be smaller than this maximum, except for NOCOLLECTION. Most of the results we show are for databases of between 5 and 10 megabytes of allocated objects.

5.2 Test Application Behavior

The test database application simulates a single process that probabilistically creates, accesses, and modifies the augmented binary trees in the database using a sequence of short transactions. The creation of garbage by the application is not controlled directly, but results from the interaction of the probability that a pointer will be overwritten and the connectivity of the database. Application transactions may disconnect and reconnect objects from the database within the context of a single transaction; collection cannot occur during the period between disconnection and reconnection.

Phases. At the highest level, the application repeatedly randomly selects one of the following operations to perform. "Create" creates a new augmented binary tree containing 1,500 objects (15 percent probability). "Visit" visits all the trees in the database, with a traversal as described below (30 percent probability). "Remove" removes 15 percent of the tree edges from a randomly selected tree in the database (50 percent probability). "No Action" performs no action (5 percent probability).

Tree Traversals. When the application decides to visit the all the trees, it probabilistically performs one of three kinds of traversals. There is a 30 percent chance of no traversal, 20 percent chance of a depth-first traversal, and 50 percent chance of a breadth-first traversal. We chose this ratio because we feel that a breadth-first style of traversal (i.e., visiting an object's siblings along with the object) is a more common access pattern than a depth-first traversal. Note that traversals only traverse the tree edges and not the dense edges.

After a type of traversal is selected, the nodes in the tree are visited probabilistically. At each edge, there is a 5 percent chance that the edge will not be traversed and thus the subtree below it not visited. Traversals are only done on the edges that constitute the binary trees, not over the dense edges.

Object Read/Write Ratio. Each time an object is visited during a traversal, there is a 1 percent chance that the data (not the pointers) in the object are modified.

Number of Collections (*Length of Run*). The number of collections is controlled by how many phases are executed in a given run. The simulation data presented here are based on runs involving approximately 25 collections. For scalability results, we varied the number of collections from 15 to 90. This number has proven to be sufficient to show significant differences among the partition selection policies.

There are other application behaviors that we need to consider, but we do not directly control them through individual parameters. Rather, the behaviors are determined by the interactions of the parameters mentioned above.

Edge Read/Write Ratio. The edge read/write ratio is a function of the phase ratios combined with the traversal and object read/write probabilities. In our simulations, the edge read/write ratio varies from about 15 to 20. We believe this read/write ratio approximates a reasonable application; with a less mutated database the issue of when to collect becomes more significant, but we do not explore that issue here.

Generating Garbage. Garbage is generated by randomly overwriting tree edges from the binary trees. All, part, or none of the subtree that the overwritten tree edge pointed to may become garbage, however, because of the presence of the dense edges.

5.3 ODBMS Design

The effectiveness of partition selection polices is more a property of the logical behavior of the database application than it is a property of the ODBMS that implements the policies. Where and when the application produces garbage, and the response of the policies to that behavior, are the key factors that must be investigated.

As a result, we make simplifying assumptions about the implementation of the ODBMS. In particular, we model a single-process application sharing a buffer with the ODBMS, which executes on the same processor. Furthermore, we assume that the collector locks the entire database during a collection and the database system does not support recovery.

I/O Buffer Size. We chose the I/O buffer size to be the same as the size of the partitions, which varied depending on the size of the simulation run. We did this because a buffer significantly smaller than a partition may cause a garbage collector to perform an excessive number of I/O operations, while a much larger buffer could overwhelm any improved reference locality that resulted from the collections. The buffer sizes range from 24 8-kilobyte pages for the smaller database simulations (which contained 4 megabytes of total allocated objects) to 100 8-kilobyte pages for the largest database simulations (which contained 38 megabytes of total allocated objects).

Object Placement. We assume each augmented binary tree is created in a breadth-first manner and the database attempts to place a new object near its parent (i.e., on the same page if possible and within the same partition). For an empty partition, then, a new tree would be placed in the database in a strict breadth-first order.

Interpartition Pointers. In our simulated ODBMS, the set of interpartition pointers are maintained in memory. We currently assume that this memory is not part of the I/O buffer. We have measured that for the test database and application described, approximately 1/4 of all pointers are interpartition pointers. As mentioned above, objects average approximately 1 pointer per 100-byte object. As a result, the fraction of memory required to maintain the interpartition pointer set is approximately 2 percent of the database size. In an actual implementation, these sets would be stored in the database. We do not know how much impact maintaining these sets in the database will have on overall performance, but in any event, all the methods we measure will incur approximately the same penalty for maintaining them.

6 Results

In this section, we show and explain the results of our simulations measuring the relative performance of the different partition selection policies and determine how sensitive that performance is to parameters of the simulation. This section is divided into two parts. In the first part, we compare the performance of the different policies in terms of throughput and space usage. We also consider the time-varying performance of secondary-memory space usage. In the second part of this section, we investigate the sensitivity of the policies to the parameters of our simulation, including partition size, collection rate, and database connectivity and size. The purpose of this second part is to better understand under what circumstances the results from the first part will hold.

6.1 General Performance Results

The tabular data reported here are means (and standard deviations, where appropriate) of 10 sets of simulation runs, each set with the same configuration parameters but with a different random seed. Only the time-varying plot in Section 6.1.3 is from a single set of simulation runs. For the data presented in Table 3 through Table 5, the partition size and buffer size were set at 48 8-kilobyte pages, resulting in between 15 and 25 partitions. The database contained a maximum of approximately 5 megabytes of live data. The connectivity of objects in the database was an average of 1.04 pointers into each object. Because we have focused on the partition selection policy and not on other policies that may affect the absolute performance of garbage collection, it is important to look only at the relative performance of the selection policies and not at their absolute performance.

6.1.1 Throughput

The performance of a database system in terms of how long it takes to complete some set of application events (creation, access, and modification) is of importance. Adding

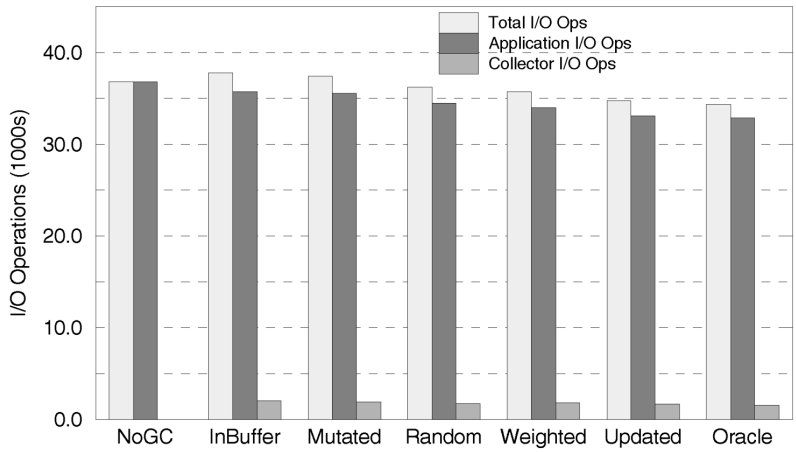


Fig. 6. Graphical view of data in Table 3.

TABLE 3
THROUGHPUT AS NUMBER OF 8-KILOBYTE PAGE I/O OPERATIONS

Selection Policy	Application I/O Ops Mean (s.d.)	Collector I/O Ops Mean (s.d.)	Total I/O Ops Mean	Relative Total I/O Ops Mean (s.d.)
NOCOLLECTION	36836 (5582)	0 (0)	36836	1.073 (0.021)
INBUFFER	35768 (6083)	2023 (207)	37791	1.099 (0.019)
MUTATEDPARTITION	35548 (5586)	1906 (236)	37454	1.092 (0.027)
RANDOM	34502 (5873)	1720 (262)	36222	1.053 (0.019)
WEIGHTEDPOINTER	33989 (5637)	1791 (315)	35780	1.041 (0.034)
UPDATEDPOINTER	33098 (5559)	1665 (207)	34763	1.011 (0.016)
ORACLE	32860 (5426)	1510 (235)	34370	1.000 (0.000)

Relative is Oracle = 1.

the burden of garbage collection—but with it the possibility of increased reference locality—further increases the importance of this measurement.

Table 3 and Fig. 6 show the throughput of the policies across simulation runs, measured as the number of page I/O operations, where fewer I/O operations indicates better performance. The figure shows that the I/O operation overhead of garbage collection is low, given the fixed collection rate chosen. (In Section 6.2.2, we show the impact of collection rate on collection I/O overhead.) While the choice of partition selection policy has a relatively small impact on collector I/O operations, it does have a larger impact on the I/O operations of the application itself. In particular, collection always lowers the number of application I/O operations, no matter which of the selection policies is used. The fact that the ORACLE policy results in the fewest I/O operations indicates that overall I/O performance is significantly improved if the goal of removing as much garbage as possible is achieved. UPDATEDPOINTER performs very close to ORACLE, and as we see in the next section, this occurs because it also reclaims the most garbage (aside from ORACLE).

6.1.2 Space Usage

Besides throughput, space usage is the other major factor in determining the quality of the database system performance. The degree to which the garbage collector finds and reclaims the most garbage determines how fast the storage space for the database will grow and how much space will be wasted. Our measure of space usage is the maximum amount of space required by the database, which includes any fragmentation, unreclaimed garbage, and empty partitions.

Table 4 and Fig. 7 show the maximum size reached by the database under each selection policy. The figure shows that the choice of partition selection policy has a significant effect on space usage. Furthermore, certain policies such as MUTATEDPARTITION and INBUFFER can result in worse overall performance than a random selection policy. As with I/O operations, UPDATEDPOINTER performs close to the ORACLE policy. The large standard deviation of WEIGHTEDPOINTER is due to the fact that its heuristic is heavily influenced by the removal of single pointers.

6.1.3 Discussion

Table 5 shows the impact of partition selection policy on garbage reclaimed and collection efficiency. The table shows that the fraction of total garbage reclaimed varies greatly, depending on the selection policy used. Furthermore, even the ORACLE policy is unable to reclaim all the garbage because it does not run frequently enough. By collection efficiency, we mean the amount of garbage reclaimed per collection I/O operation. The table shows that the efficiency of the ORACLE and UPDATEDPOINTER policies is significantly higher than any of the others. In this section, we discuss the reasons for these differences.

First, we note that INBUFFER performs worse than all the others, including RAMDOM . The reason that INBUFFER performs so badly is that the policy makes a poor assumption about the relationship between accesses and garbage generation. In particular, recall that the policy attempts to achieve the intuitively appealing goal of reducing I/O operations by collecting the partition that has the most pages in the buffer. The policy fails, however, because it assumes that objects are accessed (and thus placed in the buffer) immediately before they become garbage. We can think of two scenarios that contradict the assumption underlying INBUFFER. First, objects that are recently visited are *less* likely, not more likely, to become garbage simply because they are still being used by the application. Second, large subtrees of objects can be made into garbage by

TABLE 4
MAXIMUM STORAGE SPACE USAGE

Selection Policy	Max Storage Required: Absolute (KBytes) Mean (s.d.)	Max Storage Required: Relative Mean	# of Partitions Mean (s.d.)
NOCOLLECTION	11158 (1252)	1.53	29.5 (3.50)
INBUFFER	10229 (1344)	1.40	26.8 (3.36)
MUTATEDPARTITION	09214 (1379)	1.26	24.0 (3.59)
RANDOM	08745 (1481)	1.20	23.1 (3.67)
WEIGHTEDPOINTER	08596 (1645)	1.18	22.8 (4.24)
UPDATEDPOINTER	07721 (1387)	1.06	20.6 (3.53)
ORACLE	07298 (1262)	1.00	19.5 (3.37)

Relative is Oracle = 1.

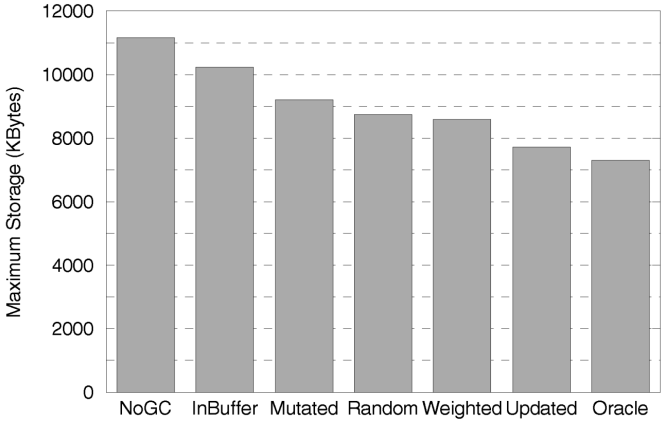


Fig. 7. Graphical view of data in Table 4.

overwriting a single root pointer, in which case there is no relation between object accesses and garbage creation. If the partition selected for collection has a large amount of live data (i.e., the assumption above fails), then collection itself will cause more I/O operations (not fewer) because all the live data in the partition will be copied. As a result, less data is collected, and during collection, more I/O operations are required. In addition, because the policy fails to select partitions containing significant amounts of garbage, it also results in a larger database size, and more application I/O operations. Because the INBUFFER policy performs worse than the others by all measures, we do not consider it further.

TABLE 5
GARBAGE RECLAIMED AND COLLECTION EFFICIENCY

Selection Policy	Amount of Garbage Reclaimed (KBytes) Mean (s.d.)	Fraction of Garbage Reclaimed (%) Mean (s.d.)	Collection Efficiency (KBytes per I/O Op) Mean	Relative Efficiency
NOCOLLECTION	0 (0)	0 (0)	0.00	0.00
INBUFFER	1498 (238)	21.48 (3.57)	0.74	0.24
MUTATEDPARTITION	2608 (355)	37.36 (5.20)	1.37	0.44
RANDOM	3104 (463)	44.52 (7.20)	1.80	0.56
WEIGHTEDPOINTER	3365 (807)	48.17 (11.56)	1.88	0.60
UPDATEDPOINTER	4293 (499)	61.62 (8.07)	2.58	0.82
ORACLE	4727 (479)	67.79 (5.35)	3.13	1.00
Actual Garbage	6999 (529)			

Relative is Oracle = 1.

Next, we note that MUTATEDPARTITION, while performing better than INBUFFER, also performs worse than RAMDOM. This performance is the result of three circumstances. First, MUTATEDPARTITION does not differentiate between initial pointer writes occurring during object creation and pointer overwrites that occur outside of object creation. Thus, it is influenced by the creation of new objects, which is not correlated to the creation of garbage. (We also considered a version of MUTATEDPARTITION that does indeed differentiate between these two kinds of overwrites. This enhancement only improved the performance of MUTATEDPARTITION to approximately that of RAMDOM). Second, MUTATEDPARTITION does not monitor creations and deletions of interpartition pointers, as the UPDATEDPOINTER policy does. For the test application, approximately 25 percent of all pointers were interpartition pointers and, as a result, MUTATEDPARTITION incorrectly attributed which partition was likely to contain garbage approximately 25 percent of the time when a pointer was overwritten. Finally, MUTATEDPARTITION fails to incorporate information gathered during the collection of a partition, as UPDATEDPOINTER does (see below).

The goal of this investigation is to develop a policy that performs better than RAMDOM and as close as possible to ORACLE. Both the UPDATEDPOINTER and WEIGHTEDPOINTER policies meet this goal. As our results show, WEIGHTEDPOINTER does not perform as well as UPDATEDPOINTER; the connectivity results presented in Section 6.2.3 explain why. UPDATEDPOINTER performs so well because the fundamental assumption it makes is an accurate one: Overwritten pointers provide substantial evidence that garbage has been created. UPDATEDPOINTER uses two mechanisms to exploit this assumption. First, by associating a count with the partition into which an overwritten pointer points, and not the partition in which the overwrite occurs, UPDATEDPOINTER more correctly relates garbage to the relevant partition. Second, and more importantly, UPDATEDPOINTER also gathers information about where garbage is likely to occur during the collection of a partition. In particular, by conceptually overwriting pointers in reclaimed garbage objects during collection, UPDATEDPOINTER propagates that information from those garbage objects to the partitions into which they point.

To provide a different perspective on the effectiveness of the policies at finding garbage, Fig. 8 shows the amount of unreclaimed garbage in the database as a function of time, where less unreclaimed garbage is, of course, better. In this graph, time is measured as the number of application events that have occurred. Note that (internal) garbage collection events, resulting from copying objects and traversing pointers, are not considered to advance time. The graph is taken from a simulation of a database whose storage grew to about 20 megabytes with no garbage collection, and to about 10 megabytes with the ORACLE partition selection policy. In the figure, one can see that the policies quickly differentiate themselves, with ORACLE and UPDATEDPOINTER doing much better than the others. RAMDOM and WEIGHTEDPOINTER stay approximately even with each other, and MUTATEDPARTITION worsens as time goes by. Toward the end, UPDATEDPOINTER and ORACLE overlap and are essentially indistinguishable.

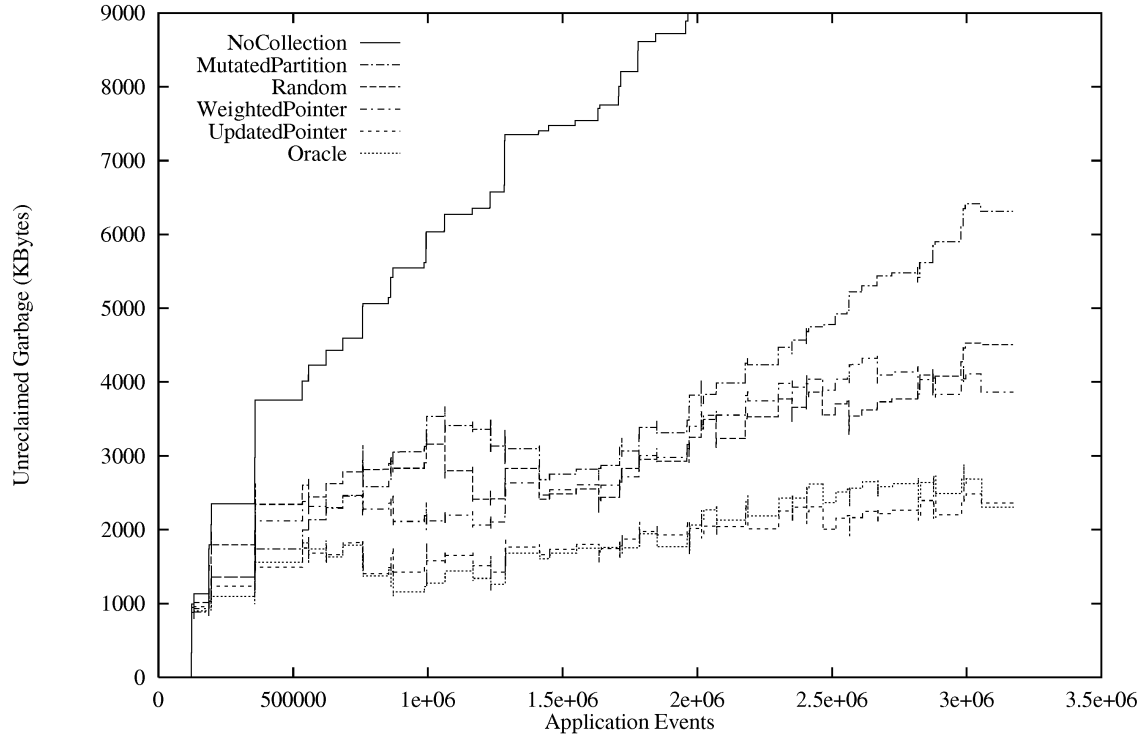


Fig. 8. Unreclaimed garbage over time.

6.2 Sensitivity of Results

The results presented so far show that the UPDATEDPOINTER partition selection policy performs close to the ORACLE policy and better than the other policies for all performance metrics in one particular configuration of the test database. We now investigate the performance of the UPDATEDPOINTER policy for a variety of different simulation parameter settings to determine if it remains close to the ORACLE policy over a range of parameter values. Together, the parameters cover the three major categories of parameters that contribute to the performance of a garbage collection algorithm. The first category is concerned with the character of the ODBMS, the second with the garbage collector itself, and the third with the application.

Each data point in Fig. 10 through Fig. 13 represents the mean of three runs. Each data point in Fig. 14 represents the mean of eight runs. Each data point in Fig. 9 and Fig. 15 represents a single run. The WEIGHTEDPOINTER, MUTATEDPARTITION, and INBUFFER policies continued their poor performance relative to the UPDATEDPOINTER policy, so to simplify the presentation of the data, we do not consider them further in this section and show only results for the NOCOLLECTION, RAMDOM, ORACLE, and UPDATEDPOINTER policies. Note that the Y-axis on Fig. 9 through Fig. 13 has been truncated to make the trends more easily identifiable.

6.2.1 ODBMS Parameters

We begin the investigation of sensitivity by considering the effect on the performance of garbage collection that results from varying parameters typically defined by the ODBMS. In particular, we investigate the effect of partition size and I/O buffer size.

Fig. 9 shows the performance (as measured by total I/O operations required) of different partition selection policies as a function of increasing both I/O buffer size and partition size. A buffer size and partition size of 1 corresponds to I/O buffers and partitions that are both 192 kilobytes in size. In order that the total amount of space considered for reclamation remains constant as the partition size changes, we also change the collection rate. So, for example, if the partition size is doubled, we must halve the collection rate.

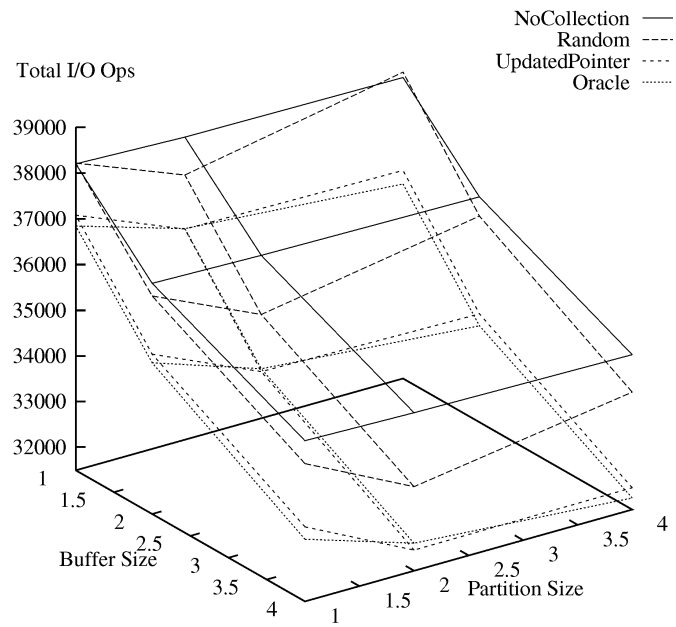


Fig. 9. I/O performance as a function of partition size and I/O buffer size.

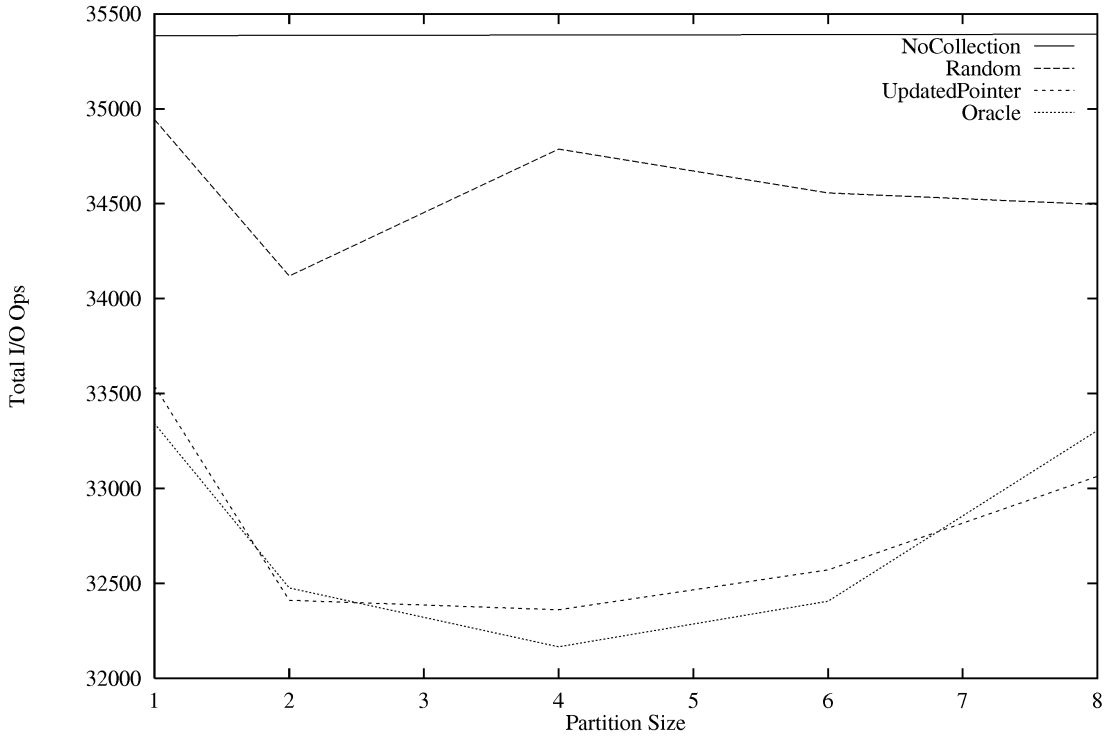


Fig. 10. Total I/O operations as a function of partition size.

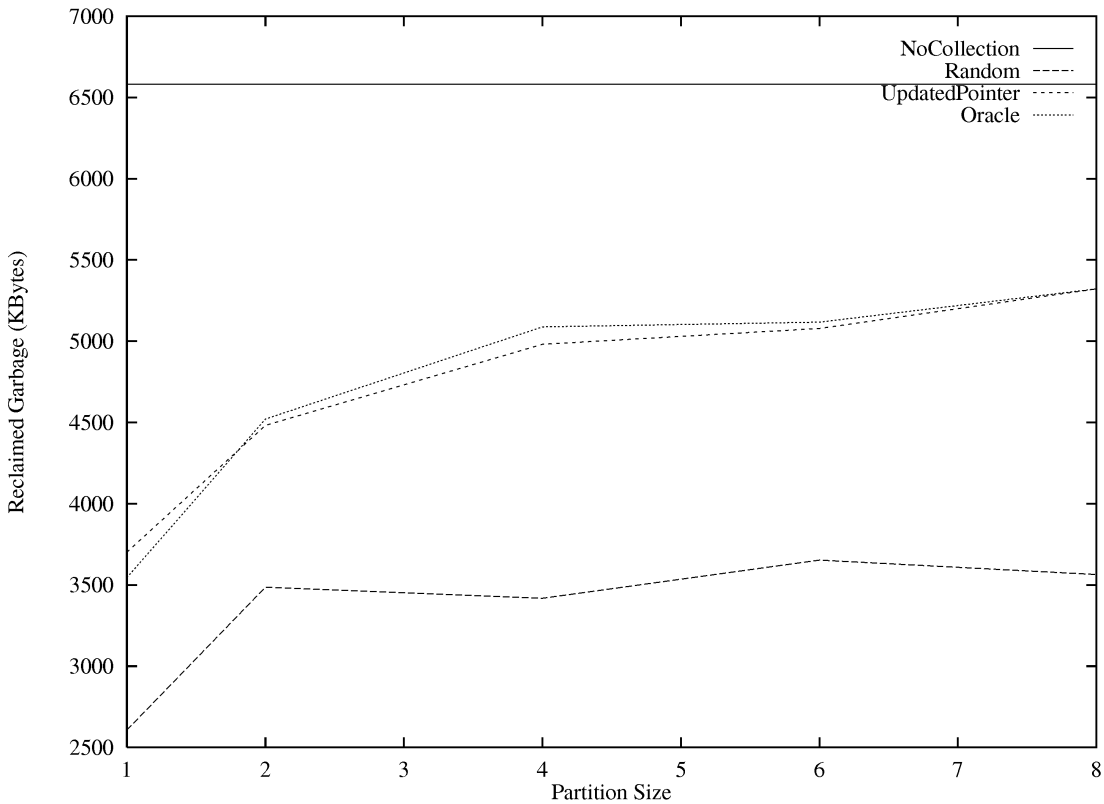


Fig. 11. Reclaimed garbage as a function of partition size.

Fig. 9 clearly shows that, as we have seen before, the UPDATEDPOINTER and ORACLE policies remain very close in performance over a wide range of parameter values. Likewise, the other policies show worse performance over the same range. Perhaps more importantly, the figure shows that, as we would expect, increasing the I/O buffer size uniformly reduces the number of I/O operations for all the policies, regardless of partition size. From this, we conclude that I/O buffer size has a performance impact on the partition selection policy that is independent of other parameters.

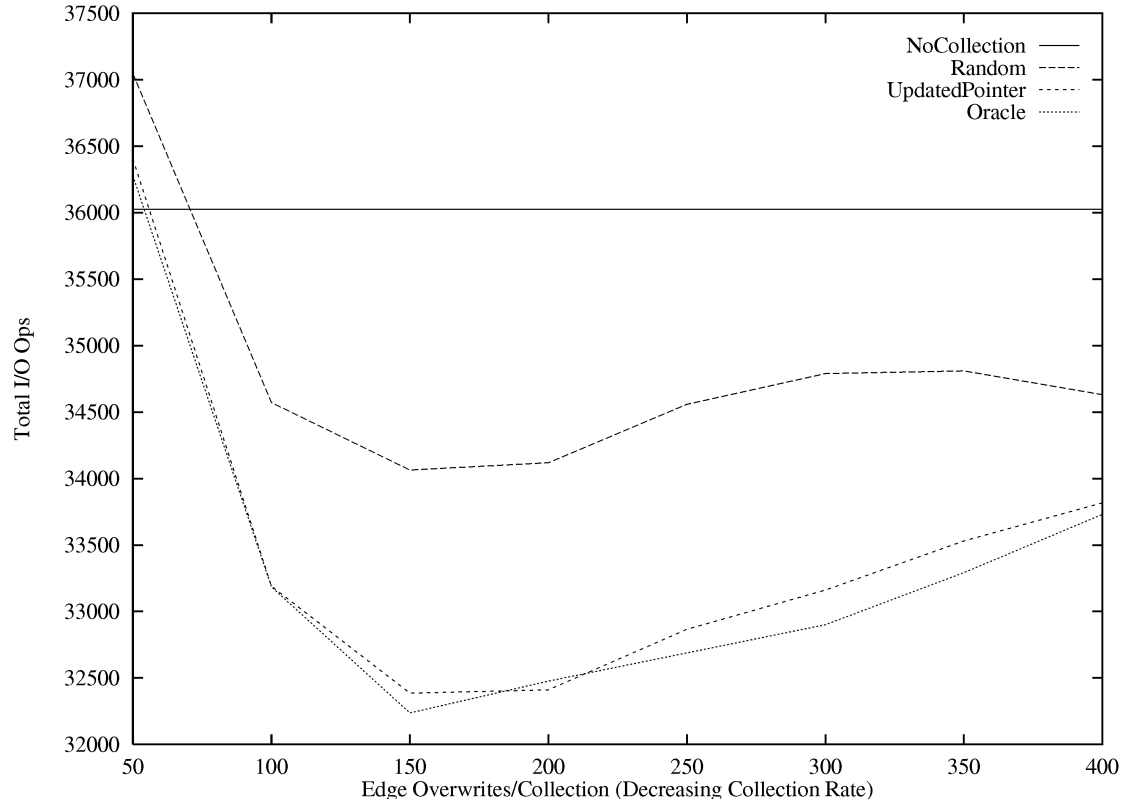


Fig. 12. Total I/O operations as a function of collection rate.

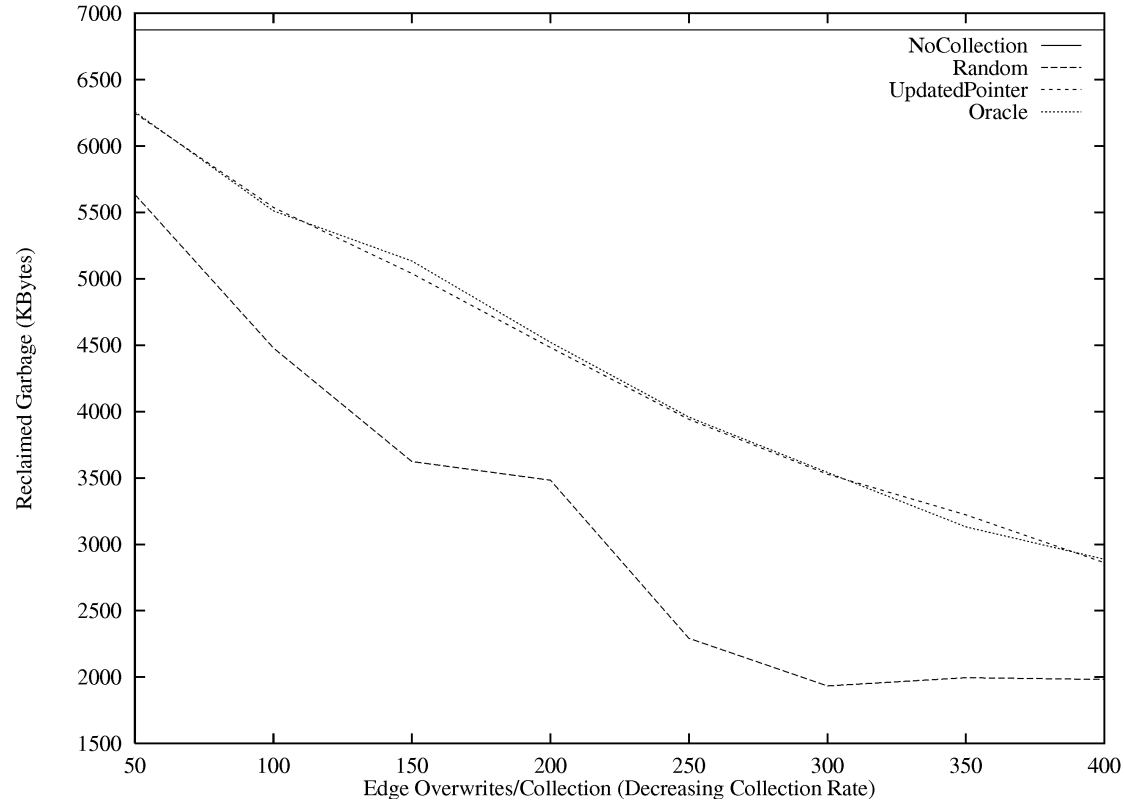


Fig. 13. Reclaimed garbage as a function of collection rate.

Therefore, designers of ODBMS garbage collection algorithms do not need to be concerned with this interaction in their designs.

To better understand the impact of partition size on collector performance, we set the I/O buffer size to a constant and investigate a range of partition sizes, essentially taking a slice through Fig. 9. In Fig. 10, we plot the number of I/O operations for the four collection policies as a function of partition size, where the I/O buffer size is held constant at 384 kilobytes. A partition size of 1 corresponds to partitions that are 192 kilobytes in size. Again, as partition size is increased, collection rate is proportionately decreased to hold the total space collected constant.

Fig. 10 more clearly illustrates behavior that is also visible in the previous surface plot. In particular, the I/O performance first improves with increasing partition size, and then degrades in both the ORACLE and UPDATEDPOINTER policies. This behavior can be explained in the following way. For large partition sizes, collections are less frequent and more garbage is retained between collections, decreasing I/O performance. As partition size goes to infinity, the I/O performance of a partitioned collector will asymptotically go to that of the NOCOLLECTION algorithm. For small partition sizes, the compaction achieved by collection is less effective because trees in the database can span multiple partitions. Also, as the size of partitions decreases, the number of interpartition pointers increases. Note that interpartition pointers from dead objects into a partition cause data in a partition to be considered alive (so-called *nepotism* in generational collectors [31]). Thus, with smaller partitions, the rate of nepotism is higher and the efficiency of garbage collection decreases.

Fig. 11 shows the amount of reclaimed garbage as a function of partition size. Again, a partition size of 1 corresponds to partitions that are 192 kilobytes in size. Here, we see clearly that even though the same amount of space is considered for reclamation independent of partition size, the smaller partition sizes result in less garbage being collected. With smaller partitions, nepotism and the related effect of distributed cyclic garbage (that is, cycles of self-referential garbage structures that cross partition boundaries) are likely to impact collector performance in terms of garbage reclaimed. Beyond a certain size, it appears that these effects are not significant. Once the partitions are large enough to contain entire binary trees from the forest, the impact of nepotism lessens.

6.2.2 Collector Parameters

We now consider an important parameter that is part of the garbage collector design, namely the rate of garbage collection. As mentioned earlier, our simulation system invokes garbage collection based on the amount of mutation performed by the application.

We vary the rate of collection to investigate the impact of this parameter on the collector performance. For these measurements, the partition size and I/O buffer size are held constant at 384 kilobytes each. Because collections are triggered after a fixed number of edge overwrites in our test database, we varied the number of overwrites between collections to control the rate parameter. Fig. 12 shows the impact of collection rate on total I/O operations required. As the number of edge overwrites per collection increases to the right, the collection rate decreases. Again, we see that the UPDATEDPOINTER and ORACLE policies perform very close to each other, while the other policies are worse. We also see that the I/O performance has a shape similar to the I/O performance in Fig. 10. This behavior can be explained in the following way. If collections occur very frequently, then little garbage will be generated between the collections, and the collector will expend wasted effort (and I/O operations) attempting to collect garbage that is not there. Likewise, if collection occurs too infrequently, then more garbage will accumulate between collections and reduce the locality of the live objects, again reducing total I/O performance. Between these extremes, the number of I/O operations reaches a minimum, which depends on a number of parameters in a complex way. We are currently investigating this relationship [13].

Fig. 13 shows the impact of collection rate on total garbage reclaimed. The line associated with the NOCOLLECTION algorithm indicates the total garbage present in the system. Here, we see that frequent collections do collect more garbage, again with the ORACLE and UPDATEDPOINTER algorithms providing similar performance. Combining the results in Fig. 12 and Fig. 13, we see that attempting to collect *all* the garbage in the database leads to degraded I/O performance. Thus, the best garbage collector will not attempt to collect all the garbage in the database.

6.2.3 Application Parameters

Finally, we investigate the performance of garbage collection as a function of parameters associated with the application, namely the average database connectivity and the database size.

6.2.3.1 Database Connectivity

We first investigate the effect of database connectivity on the performance of the UPDATEDPOINTER and WEIGHTEDPOINTER policies, specifically to investigate why UPDATEDPOINTER outperforms WEIGHTEDPOINTER in the results in Section 6.1. We use simulated databases that grow to approximately 5 megabytes. The partition and buffer sizes for these measurements were set at 48 8-kilobyte pages. The object connectivity was varied from 1.02 pointers per object to 1.66 pointers into each object. Fig. 14 shows the impact of object connectivity on the average fraction of a partition that gets reclaimed during each collection. The figure shows that as the connectivity increases, the performance of all of the policies degrades significantly due to the fact that partitions contain a smaller fraction of garbage. This behavior is explained by the observation that there is a lower probability that garbage will be created by a random pointer overwrite. Thus, all policies reclaim a smaller fraction of a partition as connectivity increases because there is less garbage to reclaim.

Note that at the lowest connectivity, WEIGHTEDPOINTER performs just as well as UPDATEDPOINTER. Given that ORACLE performs only slightly better, there is little opportunity for WEIGHTEDPOINTER to outperform UPDATEDPOINTER, even when

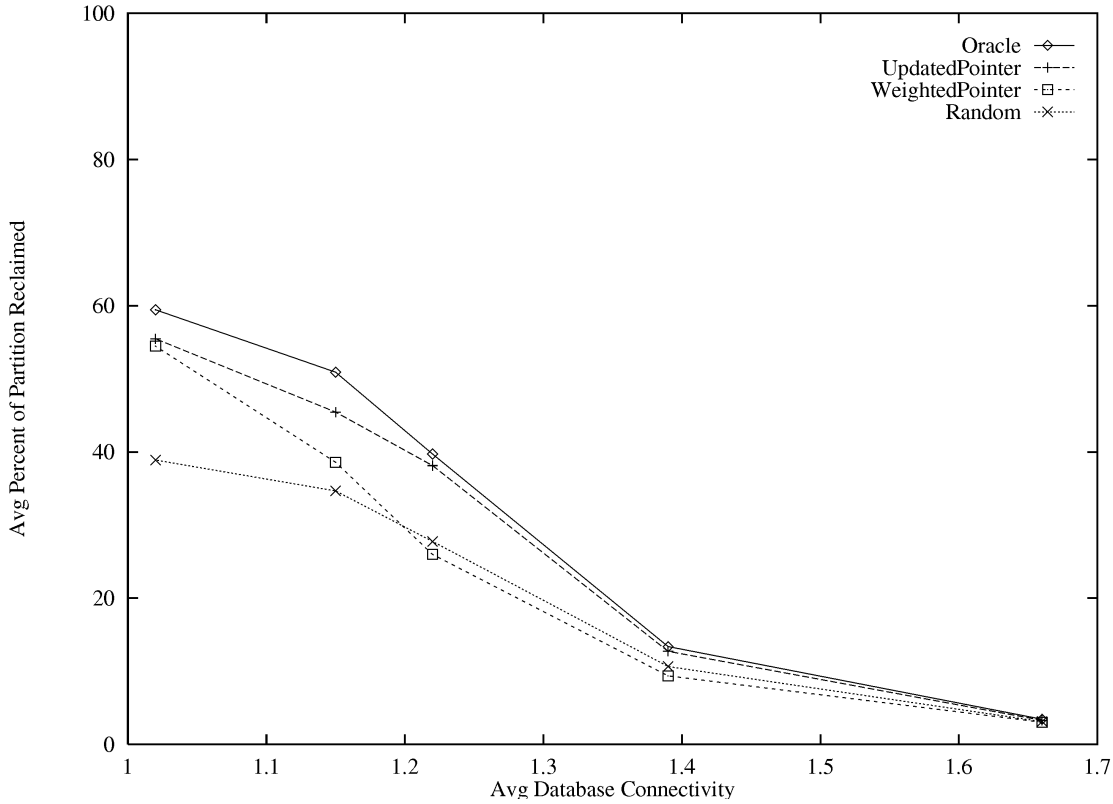


Fig. 14. Percentage of partition reclaimed as a function of database connectivity.

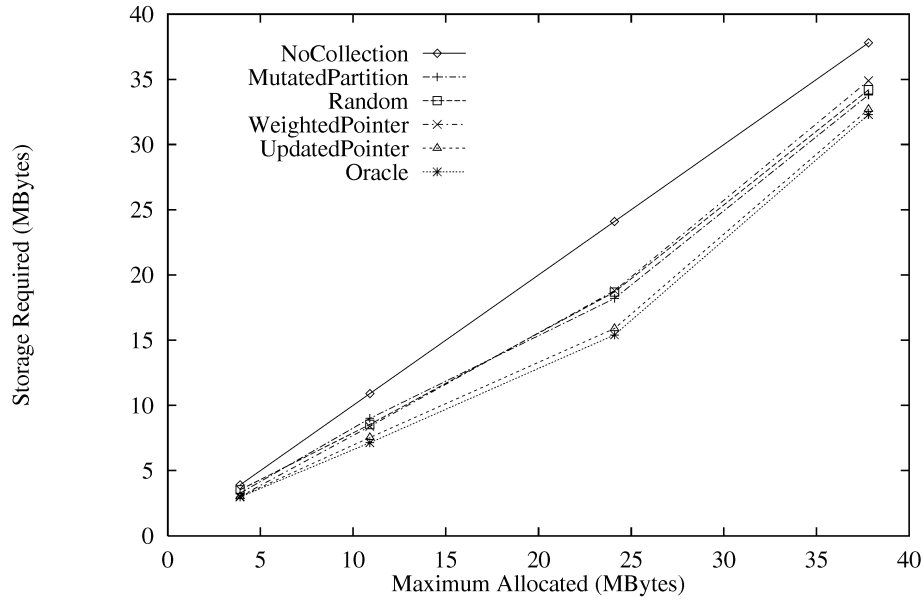


Fig. 15. Total storage required as a function of maximum allocated storage.

the database structure most closely matches its tree-structure assumption. As connectivity increases, however, and that tree-structure assumption breaks down, we see that WEIGHTEDPOINTER chooses partitions that have less garbage than the ones that UPDATEDPOINTER and ORACLE choose, and as a result a smaller fraction of the partition is reclaimed. This result is echoed in the results presented in Section 6.1, where the database connectivity was set at 1.04. Even at this relatively low connectivity, WEIGHTEDPOINTER performs slightly worse than UPDATEDPOINTER.

As the database connectivity increases, the potential for distributed cyclic garbage also increases. Our collectors do not reclaim these garbage structures, although our future work will investigate this phenomenon in more detail.

While there has been some work done in handling distributed cyclic garbage in distributed systems [17], [26], which can be applied to partitioned collection, previous work in partitioned collection has maintained that cross-partition cycles will "probably" not be a problem [33]. We have seen, however, that even small increases in connectivity can produce significant amounts of distributed cyclic garbage due to nepotism. Ultimately, we feel that reclamation of cyclic garbage will need to be performed in a graceful and scalable manner.

6.2.3.2 Database Size

We now look at the performance of the selection policies as a function of the size of the database. Partitioned collection is naturally scalable in different dimensions, with either the partition size or the number of partitions being allowed to grow. The following result explores scaling the size of the partitions as the database grows.

From the previous results, we have seen the issues involved in choosing more smaller partitions or fewer large partitions. Namely, with more partitions, collection must happen more frequently because less of the database is being collected, and the cost of maintaining the remembered sets (of interpartition pointers) becomes more expensive simply because there are more interpartition pointers. With larger partitions, a single collection will be more expensive (even if it is incremental, it will compete for buffer space), but will reclaim more space per collection.

Fig. 15 shows the total storage required for simulations of different sizes of databases, with maximum allocations ranging from about 3 to 38 megabytes. For each database size, the partition size was scaled up with the size of the database, resulting in partitions with sizes ranging from 24 to 100 8-kilobyte pages.

Fig. 15 illustrates that, as the database size increases, the relative performance of the policies remains the same. In particular, UPDATEDPOINTER remains close to ORACLE across all database sizes, while MUTATEDPARTITION does measurably worse in all cases.

7 SUMMARY

ODBMSs benefit from partitioned garbage collection because partitioning allows only a small part of a much larger database to be collected independently of the rest of the database. In this paper, we have investigated heuristics for selecting a partition to collect when a garbage collection is necessary. We have described three new partition selection policies (INBUFFER, UPDATEDPOINTER, and WEIGHTEDPOINTER), as well as one enhancement to an existing policy (MUTATEDPARTITION), for partitioned garbage collection of object databases. We compared those policies with each other and with a locally optimal, but impractical-to-implement, selection policy (ORACLE).

Using simulations based on a synthetic database, we have shown that our UPDATEDPOINTER partition selection policy performs significantly better than the other implementable policies and close to the impractical-to-implement and locally optimal ORACLE policy in all cases. We have further shown that these results hold across a range of parameter values for the ODBMS (partition size and I/O buffer size), the collector (collection rate), and the application (database connectivity and size).

The UPDATEDPOINTER policy does well because it is based on the assumption that pointer overwrites are likely to create garbage by disconnecting the object to which the pointer being overwritten points. In particular, it does better than the more naive MUTATEDPARTITION policy for the following reasons. First, it distinguishes pointer overwrites that initialize new objects, which never create garbage. Second, it recognizes when overwrites disconnect objects in partitions other than the one being modified. Finally, because it conceptually overwrites pointers from the garbage in the partitions it collects, it propagates garbage information from the partition collected to the partitions to which the garbage being collected was connected.

WEIGHTEDPOINTER, our more complex variant of UPDATEDPOINTER, is suitable only for tree-structured databases and, therefore, often performs worse than UPDATEDPOINTER in our simulations, thus not warranting its extra cost. We have seen improved performance on WEIGHTEDPOINTER when the amount of dense edges in the database is reduced. WEIGHTEDPOINTER quickly degrades, though, even with a still relatively low connectivity, as low as 1:1.

We have investigated the impact of garbage collection on object database performance over a wide variety of simulation parameters. From our investigations, we note that the impact of I/O buffer size on performance is largely independent of other simulation parameters. We investigated partition size and concluded that small partitions increase the number of interpartition pointers, and thus reduce the effectiveness of garbage collection. Large partitions, on the other hand, result in costly collections because more live data must be processed, and thus more I/O operations are incurred.

We studied the impact of collection rate on database performance. Frequent collections require many I/O operations and collect little data per collection. Infrequent collections also cause more I/O operations because the beneficial clustering effects of collection are not realized. The overall performance impact of collection rate on I/O operations can be larger than 10 percent. We have also investigated mechanisms for automatically adjusting collection rate [13].

Finally, we observed the impact of database connectivity on collection performance. From this, we conclude that with more highly interconnected databases, algorithms that detect and eliminate cyclic garbage distributed across partitions are important.

As an overall observation on the work in this area, we note that there is a general lack of understanding about the time-varying behavior of real object databases. Even existing object database benchmarks, such as OO1 [9] and OO7 [7], focus primarily on database access patterns and not on the evolution of the contents of the database. As we have noted, an understanding of algorithms for programming language system garbage collection evolved from accumulated experience and empirical results about object lifetimes in programs. With object databases, such experience and measurements do not yet exist and, as a result, many of the policy decisions mentioned in Table 1 are not well

understood. In the future, we intend to measure the time-varying behavior of real object databases.

ACKNOWLEDGMENTS

We thank the anonymous reviewers as well as Hans Boehm, Goetz Graefe, Richard Hull, Artur Klauser, Richard Snodgrass, and Alan Dearle for their helpful comments. This work was supported, in part, by the National Science Foundation under Grant No. IRI-95-21046.

REFERENCES

[1] L. Amsaleg, M. Franklin, and O. Gruber, "Efficient Incremental Garbage Collection for Client-Server Object Database Systems," *Proc. 21st VLDB Conf.*, Zurich, Switzerland, Sept. 1995.

[2] H.G. Baker Jr., "List Processing in Real Time on a Serial Computer," *Comm. ACM*, vol. 21, no. 4, pp. 280–294, Apr. 1978.

[3] P.B. Bishop, "Computer Systems with a Very Large Address Space and Garbage Collection," PhD thesis, Mass. Inst. of Technology Lab for Computer Science, Cambridge, Mass., May 1977.

[4] A. Björnerstedt, "Secondary Storage Garbage Collection for Decentralized Object-Based Systems," PhD thesis, Dept. of Computer System Sciences, Stockholm Univ., Royal Inst. of Technology and Stockholm Univ., Kista, Sweden, 1993; also appears as Systems Development and AI Lab. Report No. 77.

[5] M.H. Butler, "Storage Reclamation in Object-Oriented Database Systems," *Proc. ACM SIGMOD Int'l Conf. Management Data*, pp. 410–423, San Francisco, 1987.

[6] J. Campin and M. Atkinson, "A Persistent Store Garbage Collector with Statistical Facilities," Persistent Programming Research Report 29, Dept. of Computing Science, Univ. of Glasgow, Glasgow, Scotland, 1986.

[7] M.J. Carey, D.J. DeWitt, and J.F. Naughton, "The OO7 Benchmark," *Proc. ACM SIGMOD Int'l Conf. Management Data*, pp. 12–21, Washington, D.C., June 1993.

[8] M. Cart and J. Ferrié, "Integrating Concurrency Control into an Object-Oriented Database System," *Proc. Second Int'l Conf. Extending Database Technology*, Springer-Verlag, Mar. 1989.

[9] R.G.G. Cattell, *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 6, pp. 247–281. San Mateo, Calif.: Morgan Kaufmann, 1991.

[10] *The Object Database Standard: ODMG–93*, R.G.G. Cattell, ed. Morgan Kaufmann, 1993.

[11] H.-T. Chou and D.J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. 11th Int'l Conf. Very Large Data Bases*, pp. 127–141, Morgan Kaufmann, Aug. 1985.

[12] J. Cohen, "Garbage Collection of Linked Data Structures," *ACM Computing Surveys*, vol. 13, no. 3, pp. 341–367, Sept. 1981.

[13] J.E. Cook, A.W. Klauser, A.L. Wolf, and B.G. Zorn, "Semi-Automatic, Self-Adaptive Control of Garbage Collection Rates in Object Databases," *Proc. ACM SIGMOD Int'l Conf. Managment Data*, pp. 377–388, June 1996.

[14] J.E. Cook, A.L. Wolf, and B.G. Zorn, "The Design of a Simulation System for Persistent Object Storage Management," Technical Report CU-CS-647-93, Dept. of Computer Science, Univ. of Colorado, Boulder, Colo., Mar. 1993.

[15] D.L. Detlefs, "Concurrent, Atomic Garbage Collection," PhD thesis, Carnegie Mellon Univ., Pittsburgh, Oct. 1990.

[16] D.J. Dewitt, P. Futtersack, D. Maier, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *Proc. 16th Int'l Conf. Very Large Data Bases*, pp. 107–121. Morgan Kaufmann, Aug. 1990.

[17] A. Gupta and W.K. Fuchs, "Garbage Collection in a Distributed Object-Oriented System," *IEEE Trans. Knowledge and Data Eng.*, vol. 5, no. 2, pp. 257–265, Apr. 1993.

[18] A.L. Hosking, J. Eliot, B. Moss, and D. Stefanović, "A Comparative Performance Evaluation of Write Barrier Implementations," *ACM SIGPLAN 1992 Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pp. 92–109, Vancouver, B.C., Canada, Oct. 1992.

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

[20] E. Kolodner, B. Liskov, and W. Weihl, "Atomic Garbage Collection: Managing a Stable Heap," *Proc. ACM SIGMOD Int'l Conf. Management Data*, pp. 15–25, Portland, Ore., June 1989.

[21] E. Kolodner and W. Weihl, "Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap," *Proc. ACM SIGMOD Int'l Conf. Management Data*, pp. 177–186, Washington, D.C., June 1993.

[22] H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Comm. ACM*, vol. 26, no. 6, pp. 419–429, June 1983.

[23] D.C.J. Matthews, "Poly Manual," *SIGPLAN Notices*, vol. 20, no. 9, Sept. 1985.

[24] S. Nettles and J. O'Toole, "Real-Time Replication Garbage Collection," *SIGPLAN '93 Conf. Programming Language Design and Implementation*, pp. 217–226, Albuquerque, N.M., June 1993.

[25] K.P. Shannon and R.T. Snodgrass, "Semantic Clustering," *Proc. Fourth Int'l Workshop Persistent Object Systems*, pp. 361–374. Morgan Kaufmann, 1991.

[26] M. Shapiro, D. Plainfossé, P. Ferreira, and L. Amsaleg, "Some Key Issues in the Design of Distributed Garbage Collection and References," *Unifying Theory and Practice in Distributed Systems*, Dagstuhl, Germany, Sept. 1994.

[27] R.A. Shaw, "Empirical Analysis of a Lisp System," PhD thesis, Stanford Univ., Stanford, Calif., Feb. 1988; also appears as Technical Report CSL-TR-88-351.

[28] A.J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, Sept. 1982.

[29] M.M. Tsangaris and J.F. Naughton, "A Stochastic Approach for Clustering in Object Bases," *Proc. SIGMOD Conf. Management Data*, pp. 12–21, Denver, Colo., Mar. 1991.

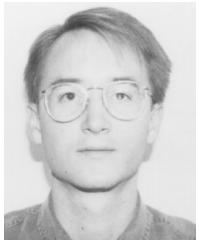
[30] D. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *SIGSOFT/SIGPLAN Practical Programming Environments Conf.*, pp. 157–167, Apr. 1984.

[31] D. Ungar and F. Jackson, "An Adaptive Tenuring Policy for Generation Scavengers," *ACM Trans. Programming Languages and Systems*, vol. 14, no. 1, pp. 1–27, Jan. 1992.

[32] P.R. Wilson, "Uniprocessor Garbage Collection Techniques," *Proc. Int'l Workshop Memory Management*, St. Malo, France, Sept. 1992.

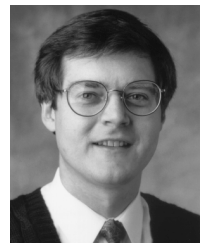
[33] V.-F. Yong, J. Naughton, and J.-B. Yu, "Storage Reclamation and Reorganization in Client-Server Persistent Object Stores," *Proc. 10th Int'l Conf. Data Eng.*, pp. 120–131, Feb. 1994.

[34] B.G. Zorn, "Comparative Performance Evaluation of Garbage Collection Algorithms," PhD thesis, Univ. of California at Berkeley, Nov. 1989; also appears as Technical Report UCB/CSD 89/544.



Jonathan E. Cook received his bachelor of science and master of science degrees in computer engineering from Case Western Reserve University in Cleveland, Ohio; and the PhD degree in computer science at the University of Colorado at Boulder. Dr. Cook is now an assistant professor at New Mexico State University. His research interests are in the areas of software process, software engineering environments, and large software system maintenance. Dr. Cook is a member of the IEEE Computer Society.

Alexander L. Wolf received the BA degree in geology and computer science from Queens College of the City University of New York, and the MS and PhD degrees in computer science from the University of Massachusetts at Amherst. He is currently on the faculty of the Department of Computer Science at the University of Colorado at Boulder. Dr. Wolf was a research member of the technical staff at AT&T Bell Laboratories in Murray Hill, New Jersey, before joining the University of Colorado. His research interests are directed toward the discovery of principles and development of technologies for supporting the engineering of large, complex software systems. Dr. Wolf has published in the areas of software engineering environments, software process, software architecture, and object databases. He is a member of the IEEE Computer Society.



Benjamin G. Zorn received a BS degree in computer science from Rensselaer Polytechnic Institute, and his MS and PhD degrees in computer science from the University of California at Berkeley. He is currently an associate professor on the faculty of the Department of Computer Science at the University of Colorado at Boulder. His research interests include the design and implementation of programming languages and object database systems. Dr. Zorn's current research projects include storage management in programming languages and object databases, profile-based optimization, empirical program and object database behavior measurement, and visual programming languages. He is a member of the IEEE Computer Society.