Partition Selection Policies in Object Database Garbage Collection

Jonathan E. Cook, Alexander L. Wolf,
and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

CU-CS-653-93 Revised December 1993

University of Colorado at Boulder

Technical Report CU-CS-653-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

Partition Selection Policies in Object Database Garbage Collection

Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

Contact Author: Benjamin G. Zorn
Telephone: (303) 492-4398
FAX: (303) 492-2844
E-mail: zorn@cs.colorado.edu

Revised December 1993

Abstract

The automatic reclamation of storage for unreferenced objects is very important in object databases. Existing language system algorithms for automatic storage reclamation have been shown to be inappropriate. In this paper, 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. Specifically, we investigate the policy that is used to select what partition in the database should be collected. The new partition selection policies that we propose and investigate are based on the intuition that the values of overwritten pointers provide good hints about where to find garbage. Using trace-driven simulation, we show that one of our policies requires less I/O to collect more garbage than any existing implementable policy and performs close to an impractical-to-implement but near-optimal policy over a wide range of database sizes and connectivities.

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 data and to support the manipulation of those data using a more powerful, programming-language-like model of computation [18]. 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 a profound impact on its performance, since such data increase the effective size of the database and can increase access time.

Designers of ODBMSs actually have some choice in how storage is reclaimed (see Figure 1).[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. 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.

Of course, 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 are ineffective, without complex enhancements, in the presence of cyclic data. Garbage collection techniques are appropriate and safe for all kinds of data, but their basis in global information makes them difficult to efficiently implement

[1] For comprehensive surveys of the field of storage reclamation, see the papers by Cohen [11] and Wilson [30].

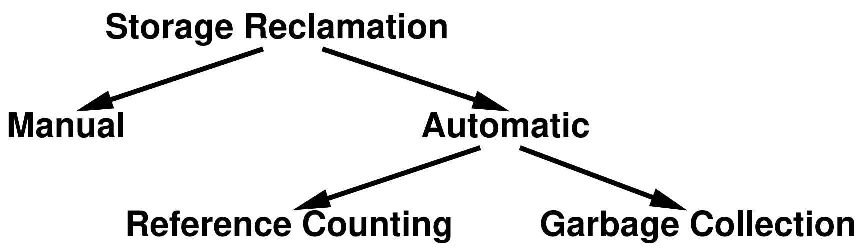

Figure 1: Approaches to Storage Reclamation.

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 [9].

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 [3, 4, 19, 20, 31]. 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 [4, 31].

This paper reports on the design and evaluation of several new storage reclamation algorithms specifically intended for use by ODBMSs. Our designs fall into the class of reclamation algorithms know as *partitioned garbage collection* algorithms [31], in which a subset of the entire database is collected incrementally and independently of the rest. The primary distinction among our algorithms is the policy by which each selects a partition (i.e., the subset) of the database to examine during a particular activation of the garbage collector. We evaluated the algorithms using trace-driven simulations [10] of applications that create, access, and modify objects. Our results show that the partition selection policy can significantly affect application performance and that a new policy we propose has the highest performance of the implementable policies that we considered.

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 [2, 21]. 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.

The obvious, but critical, 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 [14, 25, 32, 34]. Thus, these algorithms are referred to as *generational* collection algorithms [21, 28]. 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 of those criteria are: (1) partition objects based on access patterns (e.g., [24, 27]); (2) partition objects based on the unit of transfer between a server and a client(e.g., [15]); and (3) partition objects to increase locking granularity and thereby decrease overhead (e.g., [7]). We follow Yong, Naughton, and Yu in generically referring to ODBMS garbage collection algorithms that examine groups of objects independently of other groups as *partitioned* collection algorithms [31].

The previous discussion points up a key 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.

If the criterion used to partition the object space is essentially a given, then what is the crucial element in the design of an ODBMS garbage collection algorithm? The answer is *partition selection*, the selection of which partition to examine for garbage during a particular collection. Since we cannot guarantee that inaccessible objects will end up collocated in an obvious partition, as we fairly well can for programming language systems (i.e., the "youngest" partition), we must invent other ways to effectively find partitions likely to contain garbage. Thus, the essence of the problem we are addressing here is the design and evaluation of partition selection policies.

We illustrate the problem in Figure 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 entree 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 chose which single partition 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

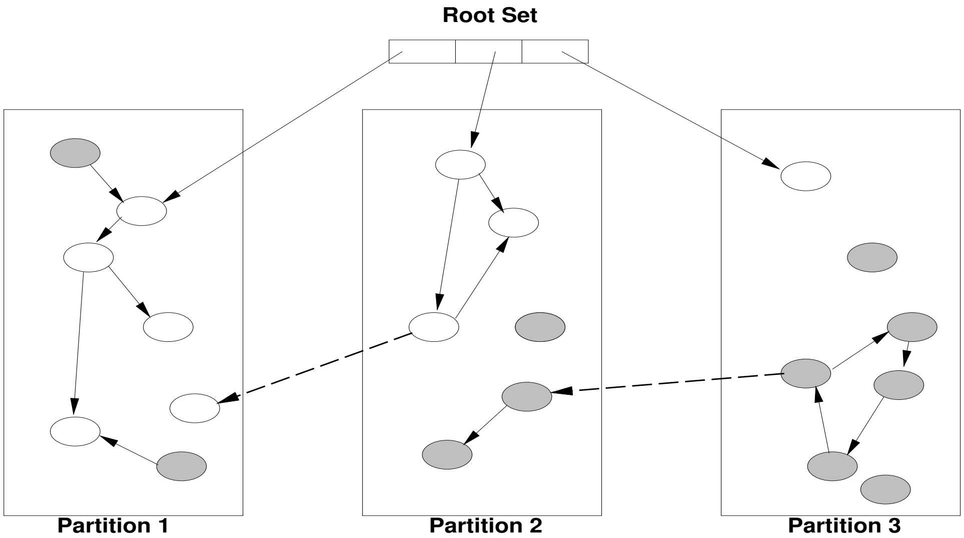

Figure 2: Example of a Partitioned Object Database

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., inter-partition 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 to be garbage. The space and time overhead of maintaining sets of inter-partition 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 because more inaccessible objects will be reclaimed.

The evaluation described in this paper compares the performance of three quite different partition selection policies, including two new ones that we devised and one previously existing one that we enhanced. All three are based on the intuition that the values of overwritten pointers provide good hints about where to find garbage. This intuition has not previously been explored, even within the programming language community. Using trace-driven simulation, we compare the policies to each other and to a near-optimal, but impractical-to-implement, policy. Our simulations show that one of the new policies we propose results in applications requiring significantly fewer I/O operations and less storage than any of the other implementable policies we considered. Furthermore, this partition selection policy has performance close to that of the near-optimal policy both in terms of total number of I/O operations performed and in terms of maximum database growth. We also show that these results hold for databases ranging from 4 to 40 megabytes in size.

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, the partitioning aspects of those algorithms have been treated secondarily at best. The first work that specifically recognized the importance of partitioning was that of Yong, Naughton, and Yu [31]. Here we briefly summarize their work and the work of others in the area of ODBMS garbage collection.

Campin and Atkinson describe a breadth-first, copying garbage collector[2] used in the PS-Algol persistent programming language [5]. 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 [22]. 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 intra-page object references are followed before any inter-page 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" [19]. 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

[2]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 non-copying algorithms leave objects in place.

Weihl propose an enhancement of their original algorithm that is also incremental and works on stock hardware [20]. As with the original algorithm, their collector accommodates system failures.

Björnerstedt describes an algorithm for collecting a distributed persistent object system [3]. 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 [4]. 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 [1] 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 [13]. Nettles and O'Toole have developed a concurrent, replicating garbage collection algorithm to support persistence in ML programs [23].

In very recent work, Yong, Naughton, and Yu present a comprehensive evaluation of incremental, reference counting, and partitioned garbage collection algorithms in client/server persistent object stores [31]. They conclude that an incremental partitioned collection algorithm shows the best performance based on a number of metrics including scalability, reclustering capability, 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 ODBMSs 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 proposed in our research more systematic, we have identified independent and

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 to maintain inter-partition pointers	• Sequential store buffer • Page or card marking
Which partition to select for collection	• MUTATEDPARTITION • UPDATEDPOINTER • WEIGHTEDPOINTER • RANDOM

Table 1: Partial List of Policies Contributing to an ODBMS Garbage Collection Algorithm. The four partition selection policies mentioned in the table are described later in this section.

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 the table). 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. In the future, we plan to study the effects of different choices for those other policies.

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 inter-partition pointers. These pointers are commonly stored in a data structure called the *remembered set* [28], which records all inter-partition pointers into a partition on a per-partition basis. Each time a write occurs there is a possibility that an inter-partition reference has been created or destroyed. The remembered set is therefore maintained by identifying when inter-partition 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 [17, 33]. In addition, most object databases already make use of the write barrier for other 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 object database is less significant than it is for language system collection algorithms, 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, MutatedPartition, UpdatedPointer, and WeightedPointer. We compare these policies with three other policies, Random, MostGarbage, and NoCollection, that are intended to place upper and lower bounds on the effect of partition selection on application performance. All six partition 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 policy 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 enhanced 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 begin again.

This policy is inexpensive to implement for the several reasons. First, it does not require any additional storage except for a 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 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.

This policy is very similar in cost to MUTATEDPARTITION. 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 UPDATEDPOINTER 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 deleted, 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.

Specifically, our implementation maintains 4 bits of weight information per object—an object's weight is the one plus the minimum of the weights of the edges pointing to it. The weight approximates the distance the object is 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. Figure 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 $2^{16-2} = 16384$.

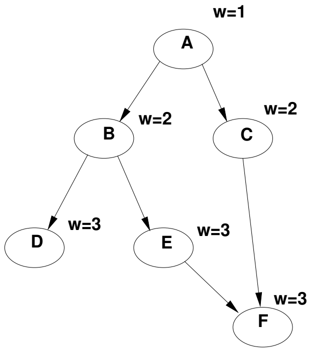

Figure 3: Object Weights in a Simple Directed Graph.

This policy is somewhat more expensive than UPDATEDPOINTER for the following reasons. First, it must maintain 4 bits of additional information (i.e., the weight) per object. Assuming objects average 100 bytes, the space overhead of the weight information is 0.5%.

Also, 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.

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.

MOSTGARBAGE. 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, MOSTGARBAGE selects the current optimal partition to collect, but that selection does not necessarily make it the globally-optimal algorithm (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 the 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 MOSTGARBAGE.

When combined with decisions for the other policies listed earlier, these six partition selection policies result in six 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. First we describe our choices for the other collection algorithm policies and then we describe the simulation techniques we used in comparing the resulting algorithms.

4.1 Complete Partitioned Garbage Collection Algorithm Design

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. 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 24–100 8-kilobyte pages per partition, depending on the size of the database used. We chose these sizes because they resulted in the 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. In our simulations, this number varied from 150–300 overwrites, which resulted in approximately 20-30 partition garbage 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 to Maintain the Inter-partition Pointers. The algorithms maintain the remembered sets for inter-partition pointers explicitly in auxiliary data structures. To understand what happens when an inter-partition pointer is created, we use the following terminology. Suppose that pointer P, pointing to partition ToPartition, is written into object O in partition FromPartition. First, we must maintain a map (the remembered set) from each partition to the inter-partition pointers into that partition; thus, the location of P is added to the remembered set for ToPartition. Second, we must maintain a map, for each partition, of all objects containing pointers that point out of the partition; thus, O is added to the out-of-partition set of pointers for FromPartition.

The out-of-partition set of objects must be 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 must 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 trace-driven simulation to evaluate alternative partition selection policies. This method has been applied effectively in evaluating many kinds of computer systems, including computer architectures [26], programming languages [29], and database systems [10, 31]. To evaluate the performance of storage reclamation in ODBMSs, our simulation system simulates the physical and logical structure of the database implementation being measured.

To perform trace-driven simulation of ODBMS collection algorithms, a trace of application events (e.g., object creations, object mutations, etc.) is generated. Details of the traces used to drive our simulations are provided in [12]. To obtain the results presented in this paper, we used synthetic, probabilistic models of application behavior to generate a test database application. Details of the ODBMS system that we model are provided in Section 5.

One advantage of using trace-driven simulation is that we are able to evaluate and compare the performance of impractical to implement, near-optimal algorithms like MostGarbage. As a result, we are able to determine how close our heuristics come to the near-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, etc. 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 ODBMSs 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. 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.

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

The characteristics of the simulated database and the operations on it can be a determining factor in the validity of the results obtained through simulation. We model a single-process application sharing a buffer with the ODBMS, which executes on the same processor. The following list presents the database characteristics and the reasons behind the choices.

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. 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. We vary the proportion of dense edges to explore how the connectivity of the database can affect garbage creation and collection, but always keep the connectivity near 1 (from 1 to about 1.2).

Object Size. Object sizes are randomly distributed around an average of 100 bytes each. 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 [4]. Yong, Naughton, and Yu use objects with an average of about 80 bytes [31]. Cattell's OO1 benchmark averages about 100 bytes per object [8]. 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 the creation of a few large objects averaging 64 kilobytes each. These are always leaf objects and comprise about 20% of the space of all objects, in a manner similar to the document nodes in the OO7 benchmark [6].

Object Placement Each augmented binary tree is created in a breadth-first manner and the database attempts to place a new object near its parent. For an empty partition, then, a new tree would be placed in the database in a strict breadth-first order.

Database Size. Our simulations vary from small databases of about 3 megabytes of allocated objects up to just over 40 megabytes; garbage collection causes the actual space used by the databases to be smaller than this maximum, except for NoCollection, of course. Most of our simulations fall between 5 megabytes and 10 megabytes of allocated objects.

Edge Read/Write Ratio. The test inputs are generated by a program that probabilisticly creates, visits, and modifies the database of augmented binary trees. Thus the edge read/write ratio is not explicitly specified but rather results from the probabilities of operations. In our simulations, the edge read/write ratio varies from about 15 to 20. 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.

Traversals. Visits to the database are performed as partial tree traversals, either in a depth-first or in a breadth-first manner. At each edge, there is a 5% 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. The particular trees that are visited are chosen randomly, with odds set at 30% for no traversal, 20% for a depth-first traversal, and 50% for 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. Additionally, when an object is visited, it has a 1% chance of being modified.

Generating Garbage. Garbage is generated by randomly deleting tree edges from the binary trees. Because of the presence of the dense edges, however, all, part, or none of the subtree that the deleted tree edge pointed to may become garbage.

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-kilobytes pages for the smaller runs (4 megabytes total allocated objects) to 100 8-kilobytes pages for the largest ones (40 megabytes total allocated objects).

Length of Run (Number of Collections). The simulation data presented here are based on runs involving approximately 25 collections. This number of collections has proven to be sufficient to distinguish significantly among the partition selection policies.

Warm-start vs. Cold-start. Our simulations have all begun with a cold start of the database—that is, starting from an empty database and 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.

Clearly, some of these choices for the design of the test database can affect the results of the simulation. But 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 database evolution modelling as appropriate for measuring garbage collection performance. Rather they are meant to measure the database system performance. 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.

6 Results

In this section, we show and explain the results of our simulations through both aggregate (tabular) data of throughput, space usage, and collection efficiency, and with time-varying plots of secondary storage space usage. 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. For the data presented in Tables 2 through 4, the partition size and buffer size was set at 48 8-kilobyte pages and the database contained a maximum of approximately 5 megabytes of live data. The time-varying plots are from a single set of simulation runs. We also show some results related to the scalability of the policies, and how the connectivity of the database affects the performance of the policies. These simulations required approximately one month of continuous execution of a MIPS R4000-based DECstation 5000/260 computer with 112 megabytes of physical memory.

Because we have focussed on the partition selection policies and not on other decisions that may affect the absolute performance of collection, such as when and how often to collect, it is important to look at the relative performance of the selection policies and not the absolute performance—further investigation into these other decisions will produce a comprehensive set of garbage collection policy decisions.

The rest of this section provides the detailed results, while section 6.6 provides a summary of the results.

6.1 Throughput

The performance of a database system in terms of how long it takes to complete some set of application events (creation, modification, and visitation) is of utmost importance. Adding the burden of garbage collection (but with it, the possibility of increased reference locality) further increases the importance of this measurement.

Table 2 shows 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. Table 2 also shows that the benefits resulting from improved locality can outweigh the costs of garbage collection. Most importantly, we see that UPDATEDPOINTER performs very close to MOSTGARBAGE, while MUTATEDPARTITION performs 9% worse on average. There is a relatively smooth progression from the best policies (UPDATEDPOINTER and MOSTGARBAGE) through to MUTATEDPARTITION, which actually shows that a poor policy can degrade performance below what NOCOLLECTION achieves (i.e, no collection can be better than bad collection).

Selection Policy	Application I/Os		Collector I/Os		Total I/Os	Relative Total I/Os	
	Mean	Std Dev	Mean	Std Dev	Mean	Mean	Std Dev
NoCollection	36836	5582	0	0	36836	1.073	0.021
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
MostGarbage	32860	5426	1510	235	34370	1.000	0.000

Table 2: Throughput as Number of Page I/O Operations (Relative is MostGarbage=1).

Selection Policy	Max Storage Required			# of Partitions	
	(kilobytes)		Relative		
	Mean	Std Dev	Mean	Mean	Std Dev
NoCollection	11158	1252	1.529	29.5	3.50
MutatedPartition	9214	1379	1.263	24.0	3.59
Random	8745	1481	1.198	23.1	3.67
WeightedPointer	8596	1645	1.178	22.8	4.24
UpdatedPointer	7721	1387	1.058	20.6	3.53
MostGarbage	7298	1262	1.000	19.5	3.37

Table 3: Maximum Storage Space Usage (Relative is MostGarbage=1).

Selection Policy	Amount of Garbage Reclaimed (kilobytes)		Fraction of Garbage Reclaimed (%)		Collector Efficiency (KB per I/O)	Relative Efficiency
	Mean	Std Dev	Mean	Std Dev	Mean	
NoCollection	0	0	0	0	0	0.00
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
MostGarbage	4727	479	67.79	5.35	3.13	1.00
Actual Garbage	6999	529				

Table 4: Collector Effectiveness and Efficiency (Relative is MostGarbage=1).

The test database size varied quite a bit between simulation runs and, as a result, the standard deviation of application I/O operations was quite high. However, the relative performance of the different policies investigated changed very little between simulation runs, and we see this fact reflected in the small standard deviation of the Relative Total I/Os.

6.2 Space Usage

Besides throughput, space usage is the other major factor in determining the performance quality of the database system. 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. We show the maximum amount of space required by the database, which includes any fragmentation and uncollected garbage, as the measure of true space usage of the database. Also, combining the amount of garbage reclaimed with the amount of work performed by the collector, where work is expressed in terms of the number of I/O operations, reveals the efficiency of the collector.

Table 3 shows the maximum size reached by the database under each selection policy; this size includes live objects, any unreclaimed garbage, and any fragmented space. Again, UPDATEDPOINTER performs close to MOSTGARBAGE, while the others are significantly and progressively worse. The ratio of 1.529 to 1 between NOCOLLECTION and MOSTGARBAGE reflects the fact that the test database has a significant amount of garbage generated in it during the simulation runs. Furthermore, the relative sizes of 1.263 for MUTATEDPARTITION and 1.058 for UPDATEDPOINTER show the importance and the impact of good partition selection.

Table 4 shows the space and collection efficiency of the policies, and supports the observation that a copying collector's efficiency is proportional to the amount of garbage it finds—that is, the efficiency is inversely proportional to the amount of live objects it must copy. This result implies that a better partition selection policy not only reclaims more garbage, but that it is *more efficient* in doing so; table 4 shows that UPDATEDPOINTER is almost twice as efficient as MUTATEDPARTITION. Also, the large standard deviation of WEIGHTEDPOINTER is due to the fact that the performance of WEIGHTEDPOINTER varies because its heuristic is heavily influenced by the removal of single edges.

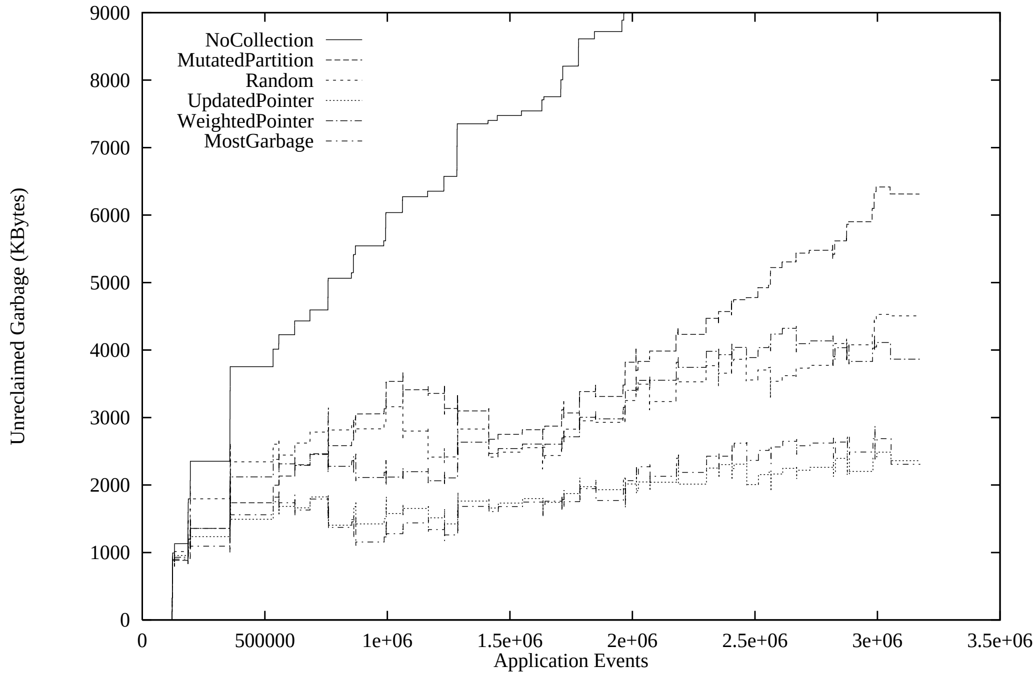

Figure 4: Uncollected Garbage Over Time.

6.3 Time-varying Performance

The tables so far have shown aggregate measurements of the performance of the various selection policies. We now show graphs of the behavior of the collectors over time during one simulation. In all these graphs, 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. These graphs are 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 MostGarbage partition selection policy.

Figure 4 shows a graph of the amount of unreclaimed garbage over time for each policy, where a lower amount shows better performance. One can see that the policies quickly differentiate themselves, with MostGarbage and UpdatedPointer doing much better than the others. Random and WeightedPointer stay approximately even with each other, and MutatedPartition worsens as time goes by. Towards the end, UpdatedPointer and MostGarbage overlap and are essentially indistinguishable.

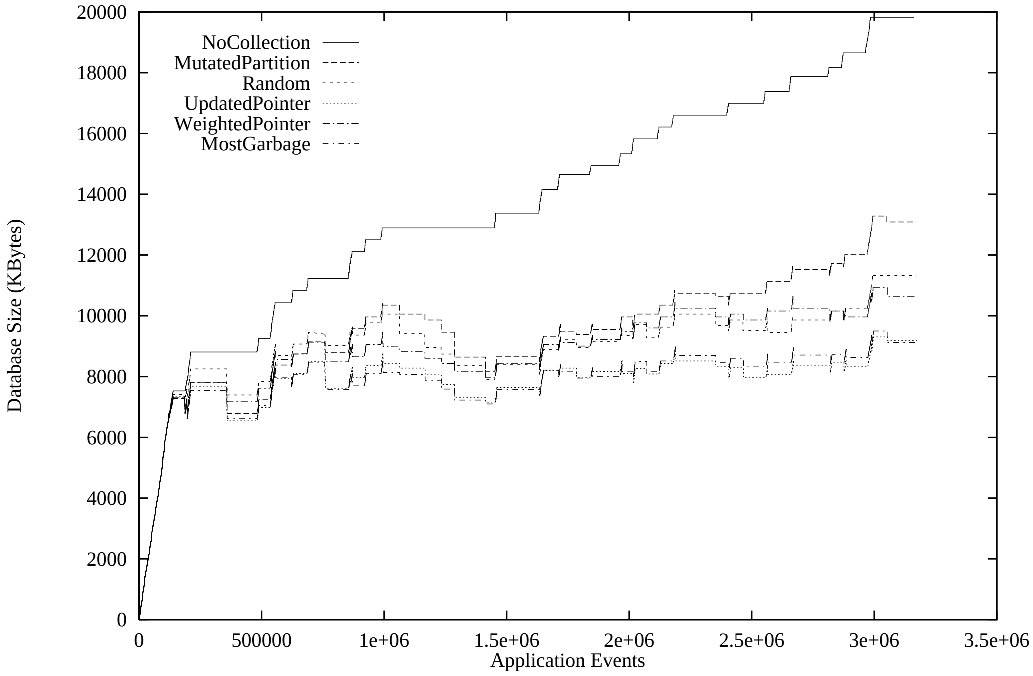

Figure 5: Database Size Over Time.

Figure 5 shows the database size, including live objects and unreclaimed garbage, over time. The separation among the various selection policies follows exactly their separation in Figure 4, with UPDATEDPOINTER following MOSTGARBAGE closely, actually dropping below (i.e., performing better than) MOSTGARBAGE for a time. As mentioned above, this is because MOSTGARBAGE is optimal in the sense of selecting the partition with the most garbage *at that instant*—it does not know what will happen in the future and, thus, might collect a partition that is going to have much more garbage created soon after the collection. One can see basically three groupings, with UPDATEDPOINTER doing as well as MOSTGARBAGE, WEIGHTEDPOINTER following RANDOM, and MUTATEDPARTITION doing poorly.

6.4 Scalability of Policies

We have looked 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 partition size being allowed

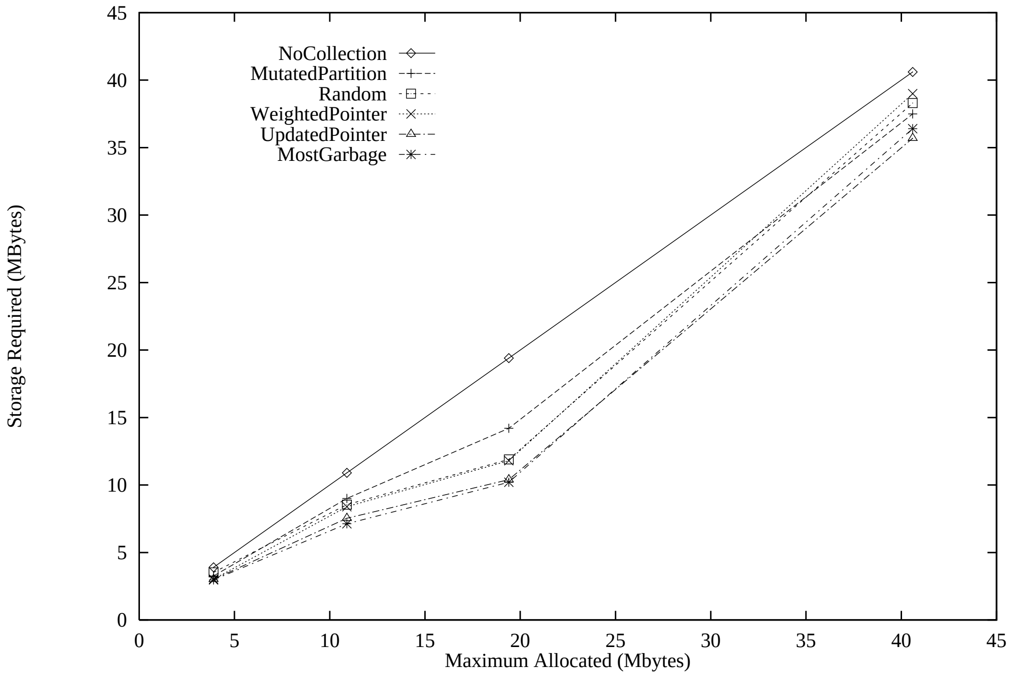

Figure 6: Storage Required as a Function of Selection Policy and Maximum Allocated Storage.

to grow with the database, or with the number of partitions being allowed to increase. Our initial investigations have explored scaling the size of the partitions.

Figure 6 shows the total storage required for simulations of different sizes of databases, with maximum allocations ranging from about 4 to 40 megabytes. For each database size, the partition size was scaled up with the size of the database, resulting in partitions with sizes ranging from 24–100 8-kilobyte pages. *[Note—the partition size of the largest database was not scaled in proportion to the others. This discrepancy is due to the fact that we were unable to rerun the large database simulations again before submitting this paper. This deficiency will be corrected in a final submission.]*.

The figure illustrates that as the database size increases, the relative performance of the most important policies remains the same. In particular, UPDATEDPOINTER remains close to MOSTGARBAGE across all database sizes, while MUTATEDPARTITION does measurably worse in all cases. The fact that the 20-megabytes data points show abnormally good collection is because the 20-megabyte (and 40-megabyte) data points are single-run values, not averages over 10 runs. So in this

Selection Policy	% of Garbage Reclaimed for Given DB Connectivity (C)			
	C = 1.167	C =1.083	C = 1.040	C = 1.005
NoCollection	0	0	0	0
MutatedPartition	28.8	35.9	38.58	39.3
Random	41.6	40.9	41.23	62.7
WeightedPointer	41.4	50.1	53.08	57.8
UpdatedPointer	57.6	61.1	62.53	74.7
MostGarbage	66.5	66.3	61.58	79.0

Table 5: Database Connectivity Effects on Garbage Collection Performance.

particular case, the 20-megabyte data points show a database that resulted in conditions favorable to collection while the 40-megabyte data points show results from a less favorable database; but in each case the best partition selection policies maintain their position.

6.5 Effects of Database Connectivity

We have also investigated the effect of database connectivity on the performance of the policies using approximately 5-megabyte simulated databases. Again, the partition and buffer size for these measurements was set at 48 8-kilobyte pages. The object connectivity was varied from 1.005 pointers per object to 1.167 pointers per object, as shown in Table 5. The table shows that as the connectivity increases, the performance of all of the policies degrades to some extent. This result can be explained by noting that as more pointers are added to the database, the number of inter-partition pointers also increases. Thus, when collecting a particular partition, the amount of data in the partition that is considered alive will increase with the number of pointers into the partition.

Note that inter-partition pointers from dead objects into a partition cause data in a partition to be considered alive (so-called *nepotism* in generational collectors [29]). Thus, as connectivity increases, so does the incidence of nepotism, and as a result, the efficiency of collection decreases.

The results in Table 5 show that nepotism plays a part (although not a large one) in reducing the efficiency of collection, even with the near-optimal MostGarbage policy. Other policies are more strongly affected; in particular, WeightedPointer, which is based on a heuristic that assumes a tree-like database, performs quite poorly with moderate amounts of interconnections.

As the database connectivity increases, the potential for distributed cyclic garbage also increases (that is, cycles of self-referential garbage structures that cross partition boundaries). 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 [16], which can be applied to partitioned collection, previous work in partitioned collection has maintained that cross-partition cycles will "probably" not be a problem [28, 31]. We have, however, seen that even small increases in the connectivity of the database can produce significant amounts of distributed garbage due to nepotism. Ultimately, we feel that distributed garbage will need to be addressed in a graceful and scalable manner.

6.6 Summary of Results

To summarize, our results show that our UpdatedPointer partition selection policy performs significantly better than the other implementable policies and close to the impractical to implement and near-optimal MostGarbage policy in all cases. These results hold across a range of database sizes and connectivities. Our complex WeightedPointer policy, most suitable for tree-like databases, most often performs worse than UpdatedPointer, thus not warranting its extra cost. We have seen improved performance on WeightedPointer when the amount of augmented pointers in the trees is reduced, this being because a tree-like database more closely approaches the weighted heuristic that WeightedPointer uses. WeightedPointer quickly degrades, though, even with a still relatively low connectivity, well below 1.1.

Furthermore, our data show that MutatedPartition performs fairly poorly and we see two main reasons this performance. First, MutatedPartition ranks the partitions based on what partitions are being modified, and not necessarily where garbage is being created. As a result, it apparently incorrectly guesses where the most garbage is. Second, MutatedPartition does not differentiate between new pointer stores 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.

An important result, supported by the data in this section, is that a copying collector is more efficient when it finds more garbage—it is actually "cheaper" to collect a partition with more garbage than it is one with less garbage. Thus, when one just looks at the amount of garbage collected, the difference between selection policies might not appear that great. However, when the *efficiency* of the collection is observed, as in Table 4, the benefit of a good policy is very clear.

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 two new partition selection policies (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 an optimal, but impractical-to-implement, selection policy (MOSTGARBAGE).

We have shown that the UPDATEDPOINTER policy, which is based on heuristically identifying garbage by observing what pointers are being overwritten, is significantly better than any existing partition selection policy. Furthermore, UPDATEDPOINTER performs close to the near-optimal MOSTGARBAGE policy in every respect over a variety of database configurations ranging from 4 to 40 megabytes in size.

Using trace-driven simulations, we have shown that the UPDATEDPOINTER policy correctly identifies partitions containing large amounts of garbage. By doing so, this policy reduces the total number of I/O operations that must be performed, reduces the total database size, and increases the efficiency of garbage collection. We believe that this policy, when combined with other incremental collection techniques, provides a solid foundation for implementing garbage collection in object databases.

Another important conclusion we have reached is 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 [8] and OO7 [6], focus primarily on the database access patterns and not the time-evolution of the contents of the database. As we have noted, an understanding of algorithms for language system garbage collection evolved from accumulated experience and empirical results about how languages allocate objects. 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.

References

[1] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[2] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, Massachusetts, May 1977.

[3] Anders Björnerstedt. *Secondary Storage Garbage Collection for Decentralized Object-Based Systems.* PhD thesis, Stockholm University, Department of Computer Systems Sciences, Royal Institute of Technology and Stockholm University, Kista, Sweden, 1993. Also appears as Systems Development and Artificial Intelligence Laboratory Report Number 77.

[4] Margaret H. Butler. Storage reclamation in object-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 410–423, San Francisco, CA, 1987.

[5] Jack Campin and Malcolm Atkinson. A persistent store garbage collector with statistical facilities. Persistent Programming Reserarch Report 29, Department of Computing Science, University of Glasgow, Glasgow, Scotland, 1986.

[6] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, DC, June 1993.

[7] M. Cart and J. Ferrié. Integrating Concurrency Control into an Object-oriented Database System. In *Proceedings of the Second International Conference on Extending Database Technology.* Springer-Verlag, March 1989.

[8] R. G. G. Cattell. *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 6, pages 247–281. Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1991.

[9] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1993.

[10] H.-T. Chou and D.J. Dewitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 127–141. Morgan Kaufmann, August 1985.

[11] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[12] Jonathan Cook, Alexander Wolf, and Benjamin Zorn. The design of a simulation system for persistent object storage management. Technical Report CU-CS-647-93, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, March 1993.

[13] David L. Detlefs. *Concurrent, Atomic Garbage Collection.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, October 1990.

[14] John DeTreville. Heap usage in the Topaz environment. Technical Report 63, Digital Equipment Corporation System Research Center, Palo Alto, CA, August 1990.

[15] D.J. Dewitt, P. Futtersack, D. Maier, and F. Velez. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 107–121. Morgan Kaufmann, August 1990.

[16] Aloke Gupta and W. Kent Fuchs. Garbage collection in a distributed object-oriented system. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):257–265, April 1993.

[17] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *ACM SIGPLAN 1992 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pages 92–109, Vancouver, British Columbia, Canada, October 1992.

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

[19] Elliot Kolodner, Barbara Liskov, and William Weihl. Atomic garbage collection: Managing a stable heap. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 15–25, Portlang, OR, June 1989.

[20] Elliot Kolodner and William Weihl. Atomic incremental garbage collection and recovery for a large stable heap. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, DC, June 1993.

[21] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[22] David C. J. Matthews. Poly manual. *SIGPLAN Notices*, 20(9), September 1985.

[23] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 217–226, Albbuqueuque, NM, June 1993.

[24] K.P. Shannon and R.T. Snodgrass. Semantic Clustering. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 361–374. Morgan Kaufmann, 1991.

[25] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, CA, February 1988. Also appears as Computer Systems Laboratory tech report CSL-TR-88-351.

[26] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[27] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In *Proceedings of the SIGMOD Conference on Management of Data*, pages 12–21, Denver, CO, March 1991.

[28] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[29] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[30] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, St. Malo, France, September 1992.

[31] Voon-Fee Yong, Jeffrey Naughton, and Jie-Bing Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proceeedings of the International Conference on Data Engineering (to appear)*, 1994.

[32] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, Berkeley, CA, November 1989. Also appears as tech report UCB/CSD 89/544.

[33] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, November 1990.

[34] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, 27(12):71–80, December 1992.