Efficient Incremental Garbage Collection for Workstation/Server Database Systems

Laurent Amsaleg
Project Rodin
INRIA Rocquencourt
Laurent.Amsaleg@inria.fr

Michael Franklin*
Dept. of Computer Science
University of Maryland
franklin@cs.umd.edu

Olivier Gruber
Project Rodin
INRIA Rocquencourt
Olivier.Gruber@inria.fr

Abstract

We describe an efficient server-based algorithm for garbage collecting object-oriented databases in a workstation/server environment. The algorithm is incremental and runs concurrently with client transactions, however, it does not hold any locks on data and does not require callbacks to clients. It is fault tolerant, but performs very little logging. The algorithm has been designed to be integrated into existing OODB systems, and therefore it works with standard implementation techniques such as two-phase locking and write-ahead-logging. In addition, it supports client-server performance optimizations such as client caching and flexible management of client buffers. We describe an implementation of the algorithm in the EXODUS storage manager and present results from an initial performance study of the implementation. These results demonstrate that the introduction of the garbage collector adds minimal overhead to client operations.

1 Introduction

A primary strength of Object Oriented Database Management Systems (OODBMS) lies in their ability to model complex objects and the inter-relationships among them. This modeling power, while allowing for a more natural representation of many real-world application domains, also raises new problems in the design of fundamental lower-level functions such as storage management and reclamation. Programming language developers have long recognized that explicit storage management (e.g., *malloc()* and *free()*) places a heavy burden on the development of large programs. Manual detection and reclamation of garbage increases code complexity and is highly error-prone, raising the risk of memory leaks and dangling pointers. For these reasons *automated* garbage collection for programming languages has been a major area of investigation [Coh81, Wil92].

The shared and persistent nature of databases, combined with the importance of efficient storage utilization provide additional motivation for the introduction of automated garbage collection. Because a database is shared among many programs and users, the knowledge of data interconnection may be distributed among many programs, making it difficult for programmers to determine

*Much of this author's work was performed as a Visiting Researcher at INRIA Rocquencourt and was partially supported by NSF Grant IRI-9409575, and the University of Maryland General Research Board.

when to explicitly reclaim storage. Also, sharing and persistence increase the potential damage caused by storage management errors, as mistakes made by one program can inadvertently affect others. These considerations argue for a very cautious approach to storage reclamation, however, neglecting to reclaim wasted space can degrade performance due to increased I/O paging.

In an OODBMS, the notion of persistence (and hence, garbage) is conceptually linked to *reachability*. The database is an object graph in which some specially designated objects serve as persistent roots. Objects that can be reached by traversing a path from a root can persist beyond the execution of the transaction that created them. Objects that are not reachable from a persistent root or from the transient program state of an on-going transaction are garbage; such objects are inaccessible and thus, the space that they occupy can be reclaimed. Reachability-based persistence provides a simple way to express persistence that is implicit and orthogonal to object type [ZM90]; therefore, some database systems, such as GemStone [BOS91] and O2 [BDK91], directly implement persistence based on reachability.[1] In addition, there has been a growing research interest in applying garbage collection techniques to OODBMS [But87, FCW89, KLW89, ONG93, ML94, YNY94]. In general, however, many existing systems still require programmers to explicitly deallocate objects. Such systems therefore, typically provide off-line utilities which must be run periodically in order to reclaim lost storage and detect dangling references. The lack of acceptance of reachability-based persistence is due at least in part to the absence of efficient implementation techniques that can be integrated with existing systems. The work described in this paper is aimed at addressing this need.

1.1 OODBMS Garbage Collection Requirements

Garbage collection has two components: *garbage detection* identifies unreachable objects and *storage reclamation* makes their storage available for reuse. It has long been recognized that traditional garbage collection techniques developed for programming languages are not directly applicable in a database context [But87, FCW89, KLW89]. OODBMS introduce several complications that can impact the correctness and/or performance of traditional garbage collection approaches. These include:

Atomic Transactions: The abort of a partially completed transaction requires that the effects of any changes made by the transaction be completely rolled-back. Roll-backs violate the property that unreachable objects must remain unreachable, and can therefore complicate garbage collection.

Fault Tolerance: OODBMS provide resilience to system failures. Garbage collection must operate in a manner that is also fault tolerant.

Concurrency: OODBMS support concurrent transactions. A garbage collector must co-exist with these transactions, and must not adversely impact their ability to execute.

Disk-Resident Data: Because much of the data managed by an OODBMS is disk resident, the garbage collector must be efficient with respect to I/O.

[1] Also, garbage collection is specified in the ODMG object database standard Smalltalk binding [Cat94].

Persistence: Modern garbage collectors for programming languages are based on the assumption that the volume of live (i.e, non-garbage) objects is small compared to the volume of garbage objects. Due to persistence, this assumption is not valid for OODBMS.

Client-Server Architecture: OODBMS typically employ a data-shipping approach in which data is cached and updated at clients. The highly dynamic, replicated and distributed nature of these updates makes client-based garbage collection difficult. Server-based garbage collection is complicated by the fact that modified data flows from clients to servers in a manner that is typically dictated by the clients; therefore, a server may at times have an inconsistent snapshot of the state of the database. The resolution of these issues is a primary focus of the work described here.

1.2 Solution Overview

In this paper we describe a garbage collector that has been specifically designed to be efficient and effective in a client-server OODBMS environment. Our collector has the following characteristics:

- It is server-based, but requires no callbacks to clients and performs only minimal synchronization with client processes.
- It works in the context of ACID transactions [GR93] with standard implementation techniques such as two-phase locking and write-ahead-logging; it requires no special hardware.
- It is incremental and non-disruptive; it holds no locks and introduces very little additional logging. It has been optimized to be efficient with respect to I/O.
- It is fault tolerant — crashes of clients or servers during garbage collection will not corrupt the state of the database.
- It can be integrated into an existing object store, requiring few if any changes to the client DBMS software. It can co-exist with performance enhancements such as inter-transaction caching [CFLS91] and flexible "steal" buffer management between clients and servers, which allows dirty pages to be sent to the server at any time during a transaction.
- It has been implemented in the client-server EXODUS storage manager, and has been shown to impose very little performance overhead.

Similar to other recent work on DBMS garbage collection [CWZ94, YNY94, ML94] we have adopted a *partitioned* approach in order to avoid the need to scan an entire database before reclaiming any space. In contrast to the other work, which advocated copying or reference counting approaches however, we have chosen to implement a non-copying Mark & Sweep algorithm. The motivation behind this choice is discussed in detail in Section 3.

The remainder of this paper is structured as follows: Section 2 describes specific problems that arise due to the client-server architecture used by most OODBMS. Section 3 compares alternative garbage collection strategies and motivates our choice of a Mark & Sweep approach. Section 4 describes our generic algorithm. Section 5 details the implementation of the generic algorithm in

EXODUS. Section 6 presents an overview of our performance studies on the implemented system. Section 7 discusses related work. Section 8 presents conclusions and future work.

2 Challenges in OODBMS Garbage Collection

In this section we focus on three specific problems that arise due to the client-server nature of OODBMS. First, however, we present a reference architecture for a client-server OODBMS.

2.1 Client-Server OODBMS Architecture

In contrast to traditional relational systems, workstation/server OODBMS architectures are typically based on a *data-shipping* approach – data items are shipped from servers to clients so that query processing (in addition to application processing) can be performed at the client workstations.

A data-shipping OODBMS consists of two types of processes that communicate via a local area network. First, each client workstation runs a Client DBMS process. This process is responsible for providing access to the database for the application(s) running at its local workstation. Server DBMS processes are the actual owners of data; they are ultimately responsible for preserving the integrity of the data and for enforcing transaction semantics.

Data-shipping systems can be structured as *page servers*, which send physical units of data between servers and clients, and *object servers*, in which clients and servers interact using logical units of data [DFMV90, CFZ94].[2] Each Client DBMS process is responsible for translating local application data requests into requests for specific database items (i.e., pages or objects) and for bringing those items into memory at the client. As a result, all of the data items required by an application are ultimately replicated and brought into the client's local memory. These items may be cached at the clients both within a transaction and across transaction boundaries. Clients perform updates on their cached copies of data items. If clients follow a steal buffer management policy with the server, then these updated items can be sent back to the server at any time and in any order, during the execution of the updating transaction. This flexibility, while providing opportunity for improved performance, also raises potential problems for garbage collection algorithms. We describe these and other problems in the following three subsections.

2.2 Problem I: Transaction Rollback and Garbage Collection

Allowing clients to send updated pages to the server at any time makes it possible that a garbage collector running at the server will encounter dirty pages. If a transaction that dirtied a page that is subsequently garbage collected needs to abort, rollback of that transaction may be difficult. An example of this problem is shown in Figure 1. In this figure (and in those of the subsequent examples) the solid black squares represent the persistent root and objects are represented by

[2] In this paper and in our implementation, we focus on page servers, however, many of the issues and solutions that are presented apply for object servers as well.

circles. Updated objects are shaded in order to distinguish between before and after states of the update operations.

In step 1 of the figure, a transaction running at the client updates object A by removing its reference to object B and then flushes a copy of the updated page to the server. In step two, the server runs the garbage collector which reclaims object B, as it is unreachable. In step three, the transaction whose update caused object B to become garbage is rolled back. If the rollback does not take garbage collection into account (as shown in the figure) then object A will contain a dangling reference to an object that no longer exists. This problem arises because the transaction

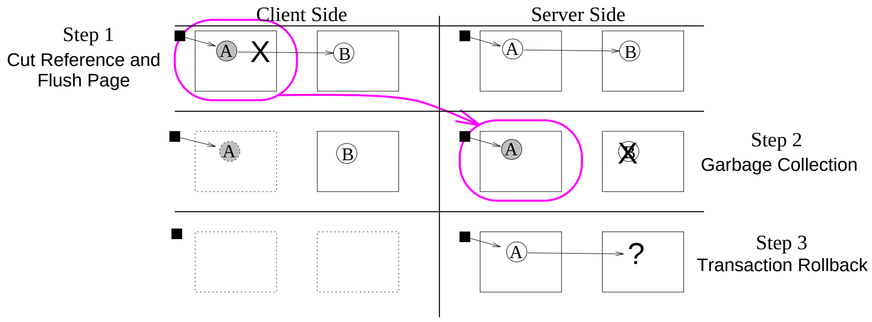

Figure 1: Rollback and Garbage Collection

rollback violates a fundamental invariant of garbage collection, namely that *unreachability is a stable property of an object*. Note that this is a non-trivial problem for rollback, as the page containing object B was never actually updated by the aborted transaction.

2.3 Problem II: Partial Flush of Multi-page Updates

Similar problems can arise even in the absence of aborts, when there are updates involving inter-object references that span page boundaries. It is possible that the garbage collector will see an inconsistent view of such updates, as there is no guarantee that all of the relevant pages will be sent to the server prior to garbage collection. There are two specific problems that can arise: 1) partial flush of updated objects, and 2) partial flush of newly created objects. Figure 2 shows an example of the first problem. In step 1, the client changes the parent of object C from object A to object B, which resides on a different page than C. The page containing C is then sent to the server, but the page containing B (the new parent) is not. In step 2 the server runs garbage collection, and reclaims object C because it appears to be unreachable. In step 3 the page containing B is sent to the server. As shown in step 4, the database is now corrupted, as object B contains a dangling reference.

The second problem that can arise due to partial flushes involves the creation of new objects, and is shown in Figure 3. In step 1, the client creates a new object C that is referenced by object A, which resides on a different page. In step two, the page containing the new object C is sent to the server, but the page containing the parent object A is not sent. In step 3, the garbage collector incorrectly determines that object C is unreachable, and reclaims it. This will result in a dangling

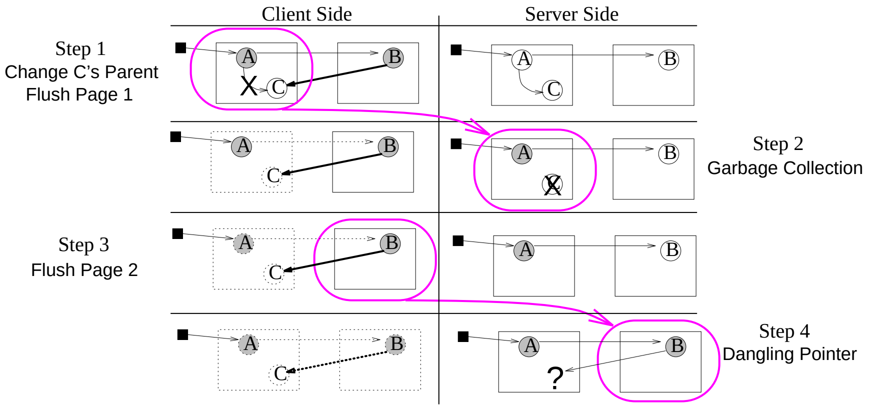

Figure 2: Incorrect Collection when Integrating Updated Objects

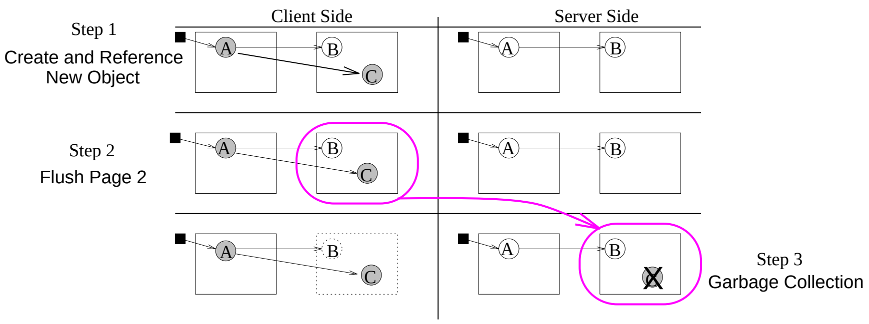

Figure 3: Incorrect Collection when Integrating New Objects

reference (as in the previous case) when the page containing `A` is sent to the server.

It is important to note that in general, it is not possible to produce an ordering of page flushes that would avoid these problems, as there can be circular dependencies among pages due to multiple updates. Furthermore, unlike the transaction rollback problem discussed previously, both of these scenarios can occur even if the garbage collector reads only committed versions of pages. Finally, it should be noted that the problems discussed in this section have analogs in incremental collection algorithms for traditional programming languages. Dijkstra and Baker have described invariants that ensure correctness for incremental collectors [DLM+78, Bak78]. In Section 4.1 we provide invariants that reflect the transactional nature of the client-server DBMS environment.

2.4 Problem III: Overwriting Collected Pages

A third set of problems involves the overwriting of a garbage collected page. One way that this can occur is during transaction REDO. If a transaction uses space on a page that was made available by the garbage collector, then if the garbage collected page is not on stable storage at the time of the crash, then the REDO of the operation may be performed on an uncollected version of the page, and hence, the REDO could fail due to lack of free space on the page. This problem is illustrated by Figure 4. Assume the maximum numbers of objects in each page is 4. In step 1, the client created 3 objects on a copy of a page that had been swept at the server but is not reflected on disk. On the

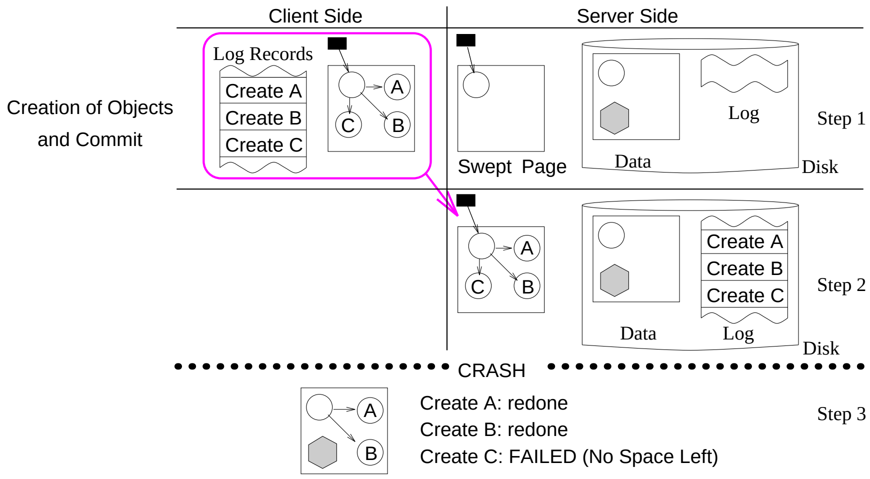

Figure 4: Redo Fails during Recovery

disk, the page still contains a garbage object that consumes space (represented by the hexagon in the figure). In step 2, the client commits, sending along the page the associated log records *which are forced to stable storage*. At the time the crash occurs, the updated page did not reach the disk yet. Recovery fails because the REDO operations are applied to the *unswept version* of the page. Log records tell the recovery to re-create 3 objects in the page. However, there is free space just for 2 of them.

A less serious instance of the overwriting problem can arise when a client updates its cached copy of a page that has been garbage collected at the server since the time that the client obtained its copy. In this case, the client will overwrite the work of the sweeper when it sends the dirty page back to the server. Unlike the previous problem, this overwriting is a matter of efficiency rather than correctness. However, note that the probability of such overwriting is increased significantly in the presence of inter-transaction caching of pages, as clients can keep pages in their caches over long time periods.

2.5 Discussion

The preceeding subsections described three specific problems that must be addressed by garbage collection algorithms for client-server database systems. These problems arise in part, due to our desire to perform garbage collection in a way that minimizes negative performance impact on existing OODBMS. Some of the problems could be solved by restricting the way in which updated pages flow from clients to servers. Moreover, none of the problems would exist if garbage collection was performed as a transaction, obtaining read and write locks and holding them until the completion of garbage collection. Neither of these solutions, however, would allow garbage collection to co-exist with transaction execution in an unobtrusive way, and both could substantially reduce the performance of a client-server OODBMS.

3 Comparing Garbage Collection Approaches

In this section we motivate our choice of a partitioned Mark & Sweep algorithm for garbage collection in client server OODBMS by briefly surveying the alternative approaches. In general, there are two families of garbage collection techniques: *Reference Counting* and *Tracing*.

3.1 Reference Counting

In reference counting systems [Col60, McB63], a count of the number of references (pointers) to each object is kept. The reference count for an object is incremented every time a new reference to it is created and is decremented every time an existing reference to it is removed. When an object's reference count drops to zero, it is known that the object is not referenced anywhere (i.e., it is unreachable) so that object can be garbage collected (which may reduce the reference counts of other objects). Reclaimed objects are inserted into a free list for future reuse.

Reference counting has two main advantages. First, garbage objects can be reclaimed as soon as they become unreachable. Second, this approach is inherently incremental; the garbage collector can be easily interleaved with transaction execution. Reference counting, however, is not suitable for garbage collecting client-server OODBMS for several reasons. First, as it is well known, reference counting schemes fail to reclaim circular structures [McB63]; garbage objects that mutually reference one another will not be collected. Second, even under deferred reference counting (e.g., [DB76]), the overhead of maintaining reference counts for the objects in a large database can be quite high [YNY94] and can require additional I/O. Furthermore, the reference count structures must be fault-tolerant, as reconstructing them as part of recovery from a system crash is prohibitively expensive.

3.2 Tracing

The other main family of collection techniques are based on Tracing. Tracing collectors consist of two basic functions. The first function is the determination of which objects are live and which objects are garbage. To make this determination, the object graph is traversed starting from known roots of persistence. All objects encountered by the trace are live, and at the end of a complete trace, all unreached objects are known to be garbage. The second function of the garbage collector is to reclaim the space occupied by garbage objects. This reclamation is accomplished either by *sweeping* the garbage objects (e.g., placing them on a free list) or by *copying* the live objects to a new area. The tradeoffs between these two approaches in an OODBMS context are the following:

Reclamation Cost: The cost of reclaiming garbage using sweeping is proportional to the number of *garbage* objects that are detected, whereas for copying, the cost is proportional to the number of *live* objects. Recall that persistence can cause the amount live data to greatly exceed the amount of garbage. Furthermore, the cost of copying objects is much higher than the cost of putting them on a freelist, particularly if the copying involves I/O or locking. Therefore, sweeping is likely to have lower reclamation cost.

Compaction and Fragmentation: Copying collectors relocate live objects and therefore, can reduce the fragmentation of the free space that is reclaimed. Sweeping schemes, however, tend to leave "holes" in the address space, as live objects are not moved. The fragmentation problems of sweeping can be largely avoided in a database context, however, through the use of *slotted pages*. Slotted pages are used in many database systems and allow items to be moved freely within a page. Thus, the space freed by the sweeper can be compacted within pages, obviating any internal fragmentation problems.

Clustering: Copying tends to group each object with an object that references it, thereby enhancing page locality. However, a copying collector will typically cluster objects in the order that they are visited (e.g., depth first search). This clustering produced by the collector may be in conflict with database requirements or user-specified hints (e.g, [BD90, GA92]). For example, using generation scavenging, objects of that differ widely in age can not be clustered together.

Fault-Tolerance: Copying causes objects to move between pages, which can greatly complicate recovery [KLW89, MRV91, YNY94]. Object movement requires the use of forwarding pointers or logical object identifiers (OIDs), both of which must be updated atomically with the object movement. Therefore, a copying collector requires a close coordination with the recovery system, and may require the acquisition of multiple write locks during object movement (as in [YNY94]). In contrast, sweeping is relatively easy to make fault-tolerant (assuming the use of slotted pages), as live objects always maintain their addressability.

These considerations have led us to adopt a sweeping-based approach to garbage collection in workstation/server OODBMS. We describe our algorithm in the following section.

4 A Partitioned Mark & Sweep Garbage Collection Algorithm

As stated in Section 1.2 we have developed a Partitioned Mark & Sweep algorithm that supports atomic transactions and is incremental, non-disruptive, and recoverable. In a partitioned collector the space to be collected (e.g., address space, database, etc.) is partitioned into separate units that can be garbage collected independently. In this section we describe the algorithm and demonstrate how it solves the problems outlined in Section 2. We first describe the algorithm in the context of a monolithic (i.e., non-partitioned) database, and then extend the algorithm to allow for independent collection of database partitions.

The algorithm is based on the following assumptions about the client-server database system:

Assumption A1: All user operations involving pointer updates are done in the context of ACID transactions [GR93].[3]

Assumption A2: The system follows the write-ahead-logging (WAL) protocol between clients and the server. That is, a client sends all log records pertaining to a page to the server before it sends a dirty copy of that page to the server.

[3] Note that other operations can be done at lower degrees of isolation if desired.

Assumption A3: Client buffering follows a "force" policy: all pages dirtied by a transaction must be *copied* to the server prior to transaction commit. The force is to the server's memory (i.e., no disk I/O is required) and clients can retain copies of the pages in their local cache.

Assumption A1 is fundamental to our algorithm, while the algorithm can be extended to tolerate the relaxation of assumptions A2 and A3 — at the cost of additional complexity and overhead. For example, the relaxation of A2 would require the garbage collector to obtain data locks; the relaxation of A3 would require additional coordination with the buffer manager to determine when certain page copies have arrived at the server. A2 must be supported by any client-server DBMS that implements WAL-based recovery (e.g., EXODUS [FZT+92], and ARIES/CSA [MN94]). A3 simplifies recovery and avoids the need for client checkpoints. The tradeoffs involved in relaxing A3 in the workstation/server DBMS environment are discussed in [FZT+92], [FCL93], and [MN94].

4.1 Algorithm Overview

A traditional Mark&Sweep [McC60] algorithm associates a "color" with each object in the object space. An object can have one of two colors: live or garbage. The colors for objects are stored in special *color maps* that are not part of the persistent object space. At the start of the collector, the all object colors are initialized to be "garbage". Mark & Sweep is a two phase algorithm. In the first (or *marking*) phase, the object graph is traversed from the root(s) of persistence. All objects encountered during this traversal are marked (i.e., colored) as live objects. In a database system (ignoring partitions for the moment), the roots of persistence are: 1) special database root objects, that are entry points into the database object graph, and 2) program variables of any *active* transactions which may contain pointers into the database. Once the entire graph has been traversed, the *sweeping* phase scans the object space, reclaiming all objects that are still marked as garbage. At the end of the sweeping phase, garbage collection is complete.

As described in Section 2, several correctness problems arise when designing an efficient, incremental garbage collector for a workstation/server DBMS. These problems stem from transaction rollbacks, partial flushes of updated pages and overwriting of collected pages. The first two problems cause the marking phase to incorrectly neglect to mark some live objects; the third problem can result in errors during transaction REDO. Given assumptions A1, A2, and A3 above, these problems can be avoided through the preservation of three invariants. The crux of our algorithm is therefore, the efficient maintenance of these invariants:

Invariant I1: When a transaction cuts a reference to an object, the object is not eligible for reclamation until the first garbage collection that is *initiated* after the completion of that transaction.

Invariant I2: Objects that are created by a transaction are not eligible for reclamation until the first garbage collection that is initiated after the completion of that transaction.

Invariant I3: Space in a page reclaimed by the sweeper can be reused only when the freeing of that space is reflected on stable storage (i.e., in the stable version of the database or in the log).

As stated previously, these invariants are similar in spirit to invariants that have been proposed for traditional incremental garbage collectors [DLM+78, Bak78]. In contrast to that earlier work, however, these invariants reflect the transactional nature of database accesses. Invariant I1 (in conjunction with assumption A1) protects against the rollback problem, as only objects made garbage by committed transactions will be eligible for reclamation. It also (in conjunction with assumption A3) protects against the partial flush of object updates, as all pages of a multi-page update will be reflected at the server when the garbage collector begins. I1 is sufficient to avoid incorrectly reclaiming live objects because the transaction that cuts the last reference to an object must have had a write lock on the reference, and therefore, it is the only transaction that can know the OID of that object. A garbage collection that begins after the completion of the transaction, will encounter all of the changes made by that transaction and therefore, will be able to correctly ascertain if the object is reachable or not. For similar reasons, Invariant I2 (along with A3) protects against the partial flush of newly created objects. Invariant I3 protects against problems that could arise due to operation REDO on swept pages after a crash, as described in Section 2.4. It ensures that REDO will reclaim any freed space that could have been lost in a crash.

4.2 Protecting Existing Objects

In order to enforce Invariant I1, we introduce a garbage collector data structure called the Pruned Reference Table (PRT). Whenever a reference to an object is cut, an entry is made in the PRT (at the server). This entry contains the OID of the referenced object and the Transaction ID (TransID) of the transaction that cut the reference. By considering the PRT as an additional root of persistence, the garbage collector is forced to be conservative with respect to uncommitted changes. That is, any object that has an entry in the PRT will be marked as live and will be traversed by the marker (if it isn't already marked live) so that its children objects will be marked as well. Therefore, a single PRT entry transitively protects all of the objects that are reachable from the protected object.

In order to make the necessary entries in the PRT, all updates to pointer fields in objects must be trapped. Traps of this form are typically implemented using a *write barrier* [Wil92, YNY94]. A write barrier detects when an assignment operation occurs and performs any bookkeeping that is required by the garbage collector. Recall that the garbage collector (and hence, the PRT) reside at the server while updates are performed on cached data copies at clients. Thus, a write barrier in the traditional sense would be highly inefficient. To solve this problem, we rely on the fact that clients follow the WAL protocol (Assumption A2). The WAL protocol ensures that log records for updates will arrive at the server prior to the arrival of the data pages that reflect the updates. At the server, the incoming log records are examined, and those that represent the cutting of a reference will cause a PRT entry to be made. Note that unlike previous work that exploits logs (e.g., [KW93, ONG93]), this algorithm processes log records as they arrive from the server — prior to their reaching stable storage.

When a transaction terminates (commits or aborts), its entries in the PRT are flagged. These flagged entries are removed prior to the start of the next garbage collection (i.e., the start of the next marking phase). The PRT entries can not be removed exactly at the time of the transaction termination even though all dirtied pages are copied to the server on commit. This is because an on-going marker may have already scanned the relevant parts of the object graph using the previous copies of the objects. The next time the marker runs, however, it is known that it will see the effects all of the updates made by the committed transaction, so the PRT entries for that transaction can be removed.

The PRT does not have to be managed in a recoverable fashion; in the event of a server crash during a garbage collection, the garbage collector will be restarted from scratch after recovery is complete. In this case, the old PRT entries are not needed because for each entry the corresponding transaction has either aborted or committed. If aborted, its effects have been removed from the database and thus, will not be encountered by the marker when it is restarted; if committed, its effects are reflected in the database at the server, and will thus be encountered by the marker when it is restarted.

4.3 Protecting New Objects

While the PRT mechanism solves the problems raised for existing objects, it does not solve the problem involving the partial flush of newly created objects. We therefore, introduce a similar structure called the Created Object Table (COT). As with the PRT, the COT is maintained at the server and is updated based on log records received from the clients. When a log record reflecting the creation of an object arrives at the server, an entry is made in the COT. This entry contains the OID of the new object and the TransID of the transaction that created it. In contrast to the PRT which is used during marking, the COT is used during the sweeping phase of garbage collection.

The sweeping phase linearly scans the pages that constitute the object space. For each page, it reclaims any objects that have been left colored as garbage by the marker. For each page, the sweeper checks to see if there are entries in the COT for any of the objects on that page, and if so, it does not reclaim those objects, regardless of their color.[4] Note that the sweeper does not need to traverse newly created objects. This is because the objects referenced by a newly created object can only be: 1) other new objects (which are protected by the COT), 2) existing objects that are also referenced elsewhere (which are considered live by the marker), or 3) existing objects that are referenced solely by this new object. An object in this last category must have had a different reference pruned by the same transaction that created the new object (because the new object must have been write locked) and therefore, will be protected by the PRT.

As with the PRT, entries in the COT are flagged for removal when the transaction that created them terminates and the flagged entries are removed at the beginning of the next garbage collection

[4]An optimization to reduce the size of the COT is to make a single entry for an entire page if many objects are created on that page. In this case the sweeper will just ignore the entire page.

cycle. The reasons for delaying removal of these entries is also similar. Also, like the PRT, the COT does not need to be managed in a recoverable fashion.

4.4 Preventing Overwrite of Collected Pages

Invariant I3 states that the garbage collector must ensure that the effects of a page sweep are reflected on stable storage. This can be done by having the sweeper write a small log record for every page that it modifies. Typically, the sweeper reclaims the space occupied by a garbage object using the space management mechanism of the slotted pages. The color map of the page, therefore, can be used to create a *logical* log record for the sweep. Color maps contain a single bit for every object in the page, and therefore are quite small. Furthermore, no before image data is logged, as page sweeps never need to be undone.[5]

Logging protects the system from problems that can arise during transaction REDO (as described in Section 2.4), however, it does not protect from sweeper work being wasted due to page overwrites. This problem can be reduced in two ways. First, the sweeper can try to obtain a "no-wait" *instant* read lock on a page. Such a lock is released immediately upon being granted, and the sweeper does not block if the lock is not granted. If the lock is not granted, then some transaction has a write lock on the page. In this case, the sweeper simply skips the page, as any work that it does will only be overwritten when the transaction holding the lock commits. Secondly, as described in Section 2.4, the overwrite problem is exacerbated in systems that perform inter-transaction client caching. For such systems it is possible to have the sweeper exploit the cache consistency mechanism in order to reduce the potential for clients to update unswept page copies. Once such mechanism is described in Section 5.

4.5 The Marking and Sweeping Phases

Our algorithm simply extends a traditional Mark & Sweep algorithm with the techniques described in the preceeding sections. At the start of a garbage collection, a color map is initialized for each page in the object space. Color maps contain a bit for each possible object on the page (i.e. the maximum number of slots in a page) and all bits are initially set to the "garbage" color. The marking phase traverses the object space starting from the database root of persistence. When an object is encountered, then the color map is checked and if the object is already marked "live" then the traversal is pruned at this point. Otherwise the bit is set to "live" and the OIDs that appear as references in the object are added to a list of objects to be traversed. When the traversal from the persistent root is completed, the entries in the PRT are then traversed in the same manner. The PRT allows the marker to be run incrementally, as it supports the arbitrary interleaving of client transaction updates and marking operations. *It is important to note that because the PRT and the COT protect objects for the duration of any transaction that could cause that object to be reclaimed, the garbage collector can ignore the program state (i.e., program variables, stacks, etc.)*

[5] As described in Section 5, logging can be even further reduced if media recovery is not to be supported.

of any transactions that are concurrently executing at the clients. This significantly reduces garbage collector overhead and complexity.

When the marker completes its traversal, the sweeper phase is started. The sweeper scans the pages linearly in order to maximize I/O bandwidth. Before reading a page from disk, the sweeper first checks the color map for the page. If the page has no garbage objects on it, then the sweeper moves on to the next page. If the page has no live objects on it, then the sweeper deallocates the page — without reading it in from disk. Finally, as described in Section 4.4, the sweeper attempts to get an instant read lock on the page, and if unsuccessful, it simply moves to the next page. If the sweeper does need to sweep a page, then it reads the page into memory, modifies the color map so that all objects that have entries in the COT are marked live, logs the color map, and then frees the objects that are not marked live.[6] The sweeper can be run incrementally, with the sweep of a single page being the finest granule of operation. The WAL rule (assumption A2) ensures that the sweeper will see all relevant COT entries for the pages it sweeps.

Note that both phases of the garbage collection are fault tolerant — if the system crashes at any time during a garbage collection cycle, then upon recovery, the crashed garbage collection is simply abandoned, and a new one can be started from the beginning. If the system crashes during the marking phase, no changes have been made to data pages, so there is no danger of corrupting the database, and no work to be done for recovery. When the garbage collector is restarted, it will begin by initializing all of the color maps. The contents of the PRT at the time of the crash will be lost, but they are not needed, as after recovery the server has a consistent snapshot of the database.

If the system crashes during the sweeping phase, then some swept pages will have made it out to disk and some will not have. Swept pages are internally consistent so there is no undo work for those pages that are on disk. Pages that had not made it out to disk can simply be ignored — the next garbage collection will sweep them. As with the PRT, the COT entries that existed before the crash will not be needed by the garbage collection that is started after recovery.

4.6 Collecting Partitions

In this section we briefly discuss how to extend the monolithic Mark & Sweep algorithm described in the previous sections to allow independent garbage collection of disjoint partitions of the object space. We assume that partitions are sets of pages. The actual partitioning of the object space can be done according to physical considerations (e.g., file extents) or logical considerations (e.g., by class or relation). Partitions must be disjoint (i.e., each object belongs to exactly one partition), however, objects may reference each other across partition boundaries. In order to allow for partitions to be collected independently, each partition must have a separate persistent root object. In addition, each partition must have an associated list of incoming references that originate from other partitions. This list is referred to as the *Incoming-Reference List* (IRL). Conceptually, the IRL contains the OID of the (local) destination object and the ID of the partition in which the

[6] A latch must be held on the page while the objects are being freed in order to avoid the arrival of a new page during this operation.

(foreign) source object resides for each such reference. The IRL serves as an additional root of persistence for the partition-local collector. Options for deciding when a partition should be collected have been studied in [CWZ94].

Figure 5 shows an example of two partitions containing objects with cross-partition references. Note that the objects themselves point directly to each other and do not involve the inter-partition reference list. Similar schemes are often used by distributed garbage collection algorithms to handle inter-node references (e.g.,[SGP90, ML94]).

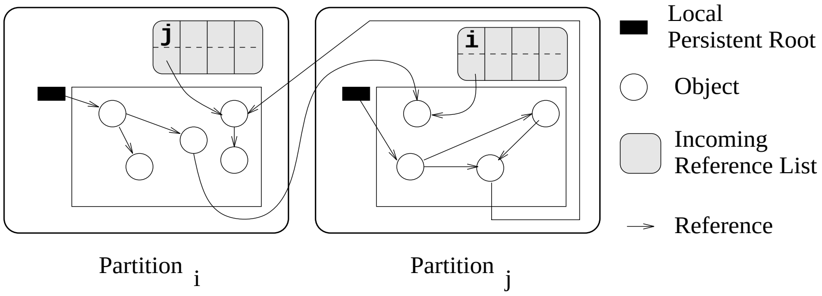

Figure 5: Partitions and Incoming Reference Lists

The IRL mechanism is transparent to programmers, and therefore, updates that may require IRL modifications must be trapped. This is handled in the same manner as the PRT and COT: the server examines incoming log records. When an update that creates a cross-partition reference is detected, an entry is made in the appropriate IRL. The removal of IRL entries is performed by the garbage collector. The marking phase traverses a partition from the persistent roots (including the IRL); when it encounters a reference to an object in a different partition, it marks the corresponding entry in the remote IRL and stops traversing that path. At the end of the marking phase, any unmarked remote IRL entries originating from the current partition can be removed, provided that the transaction that created the entry has committed (for reasons similar to the flagging of PRT and COT entries).

There are two drawbacks to this approach. First, in contrast to the PRT and the COT, IRLs have to be managed in a fault-tolerant way. Upon recovery all IRLs must be restored to their state at the time of the crash; otherwise, some remotely referenced objects may be collected erroneously. Thus, IRLs must be maintained in database pages (rather than with in-memory structures), and updates to them must be logged. Secondly, as is well known, this type of approach can not collect cycles of garbage that are distributed across multiple partitions. Separate algorithms such as Hughes' inter-partition collector [Hug85] can be used to collect such cycles periodically. In general, however, the partitioning of the object space should be done in a way that minimizes the number such references.

5 Implementing the Garbage Collector

As stated in Section 1, the design goals for garbage collection include: 1) it should impose minimal overhead on client transactions, 2) it should be efficient and effective in collecting garbage, and 3) it should be relatively straightforward to integrate the collector in existing client-server database systems. In order to assess our algorithm in light of these requirements we have implemented it in the client-server version of the EXODUS storage manager[FC92, Exo93]. In this section we describe the implementation of a single-partition collector. The extensions for multiple partitions with inter-partition references discussed in Section 4.6 are currently being added.

5.1 The EXODUS Storage Manager

Our initial implementation is based is the EXODUS storage manager v3.1 [Exo93]. EXODUS is a client-server, multi-user system which runs on many Unix platforms. It has been shown to have performance that is competitive with existing commercial OODBMS [CDN93]. EXODUS supports the transactional management of *untyped* objects of variable length, and has full support for indexing, concurrency control, recovery, multiple clients and multiple servers. Data is locked using a strict two-phase locking protocol at the page or coarser granularity. Recovery is provided by an ARIES-based [MHL+92] WAL protocol [FZT+92]. EXODUS extends a traditional slotted page structure to support objects of arbitrary length[CDRS86]. "Small" data items (those that are smaller than a page) and the headers of larger ones are stored on slotted pages. Internally, objects are identified using physical OIDs that allow pages to be reorganized without changing the identifiers of objects.

EXODUS is a page server; updates are made to the local copies and the resulting log records are grouped into pages and sent asynchronously to the server responsible for the updated pages. Dirty pages can be sent back to the server at any time during the execution of a transaction; a WAL protocol ensures that all necessary log records arrive at the server before the relevant data page. At commit time, copies of any remaining dirty pages are sent to the server. The client retains the contents of its cache across transaction boundaries, but no locks are held on those pages. Cache consistency is maintained using a check-on-access policy (based on "Caching 2PL" [CFLS91]). For recovery purposes all pages are tagged with a Log Sequence Number (LSN) which serves as a timestamp for the page. The server keeps a small list of the current LSNs for pages that have been recently requested by clients. When a client requests a lock from the server, the server checks its table to see if it can determine that the client has an up-to-date copy of the page; if not, the server sends a copy of the page back to the client along with the lock grant message.

To summarize, the EXODUS storage manager supports the three fundamental assumptions on which the garbage collector depends in terms of concurrency and transactions (see Section 4), and also has desirable properties such as slotted pages. In addition, the EXODUS server uses non-preemptive threads, which simplifies the implementation of the garbage collector. However, the system also provides features that present challenges for garbage collection, such as client caching, a steal policy between clients and servers, asynchronous interactions between clients and servers, a streamlined recovery system, and optimizations to avoid logging in certain cases.

5.2 Implementation Overview

The current implementation of the garbage collector in EXODUS is primarily a proof-of-concept implementation. It is well integrated with concurrency control and recovery and has been heavily tested; including its fault tolerant aspects. However, there are some limitations of the current implementation. First, as stated above, the framework is in place to support multiple partitions (we currently use an EXODUS "volume" as a partition) and do much of the checking that is needed to manage IRLs, but the IRL scheme is not yet implemented. Secondly, we currently collect only small-format objects (i.e, those that are smaller than 8K bytes). The extension to large-format objects, is straightforward but was not necessary for our purposes. Finally, because the Exodus storage manager does not know the types of the objects that it stores, we store a bitmap in the initial bytes of the data portion of each object. The bitmap indicates which OID-sized ranges of bytes in the object contain actual OIDs and is used by the marker during its traversal. These bitmaps are created automatically when objects are allocated using a C++ constructor.

As stated previously, the server scans incoming log records to determine if the logged update requires any entries to be made in the garbage collector data structures. In particular, the server makes an entry in the COT for any `CREATE_OBJECT` log record, and makes an entry in the PRT for any `MODIFY_OBJECT` log record that involves overwriting an OID. To facilitate this check the client sets a flag in the log record header that tells the server that it should examine the record. In the current implementation, if an operation updates multiple pointers, the client generates a separate log record for every pointer update. This limitation could easily be removed, if desired, by logging bitmaps or adding information to log records.

In order to add garbage collection to the EXODUS server, we created a new type of server thread called the `gcThread`. When a collection starts on a partition a new `gcThread` is spawned. The `gcThread` initializes the garbage collector data structures, runs the marker phase and then runs the sweeper phase. At the end of the sweeper phase, the `gcThread` terminates. As stated previously, the marker and sweeper both run incrementally. When the `gcThread` gets the processor, it starts a timer (currently set at 50 msec). If the timer expires during the marker, then the marker finishes examining the current object and then gives up the processor. If it expires during the sweeper, then it finishes sweeping the current page and then gives up the processor. The `gcThread` is woken up when there are no outstanding client requests that need the processor.

The implementation required approximately 4000 lines of new or modified code on the server-side; the bulk of which was for the `gcThread` itself. The client-side required only 200-300 lines of new or modified code. The algorithm was implemented in EXODUS with only minor changes. Three implementation issues, however, deserve mention. First, to help reduce the potential for clients to overwrite the work of the sweeper we exploit the cache consistency mechanism. The sweeper updates the sequence number on each page that it modifies. Any transaction that subsequently tries to lock such a page will then receive the swept copy even if they already have a cached copy of the page. To avoid recovery problems, however, the sequence number placed on the page by the sweeper must be guaranteed to be lower than the sequence number that will be placed on the page by any subsequent client update.

Second, EXODUS has no general support for allowing non-recovery-related server threads to create log records. Although there are workarounds, we chose to avoid logging completely in the `gcThread`. We accomplish this by setting a flag when a page is swept; the flag is cleared when the page is scheduled to be written to disk. If a client obtains a page that has the flag set, then it logs the slot allocation information (from the page header) prior to performing any updates on a page. This log record is only generated by the first transaction to update such a swept page.

The third issue results from an optimization that EXODUS uses to reduce logging during bulk loading and other create-intensive operations. When a new page is allocated to hold new objects, the individual object creations are not logged; rather, the entire page is logged when it is copied back to the server. This optimization, while providing better performance for EXODUS, deprives the garbage collector of the information on individual object creations. As a result, in the EXODUS implementation of the collector, we enter page IDs in the COT rather than individual OIDs. When the sweeper encounters a page that has an entry in the COT, it simply skips that page. Furthermore, when a newly allocated page arrives at the server, the server must scan all of the objects on the a page to determine if any new IRL entries are required.

6 Performance Measurements

In this section we describe an initial study of the performance of the implementation of our garbage collection algorithm in EXODUS. For all of the performance experiments described in this section, the EXODUS server was run on a SPARCstation LX with 32MB of main memory. The log and the database were stored on separate disks, and raw partitions were used in order to avoid operating system buffering. The page size for both data pages and log pages was set to 8KB. All times were obtained using *gettimeofday()* and *getrusage()*.

In order to gain an understanding of the collector's performance we created artificial data partitions that allowed us to vary specific factors that can impact that performance. All of the data partitions used in the study consist of simple linked-lists of objects. Each object is 80 bytes long, and along with EXODUS page and object headers, 84 objects can fit on a page. The objects are allocated in contiguous pages in an EXODUS file. The pages are fully packed with objects, however, we vary the percentage of garbage objects as an experimental parameter. As shown in Figure 6 we also vary the clustering of objects in pages. The clustering factor determines the percentage of the objects on a page that the marker can scan with out crossing a page boundary. This factor impacts the performance of the marking phase of the garbage collector; the sweeping phase scans pages linearly so it is not impacted by this factor.

This study examines three different aspects of the implementation's performance: 1) the overhead added to normal client processing by garbage collection, 2) the stand-alone performance of the collector, and 3) the performance of garbage collection and client transactions running concurrently.

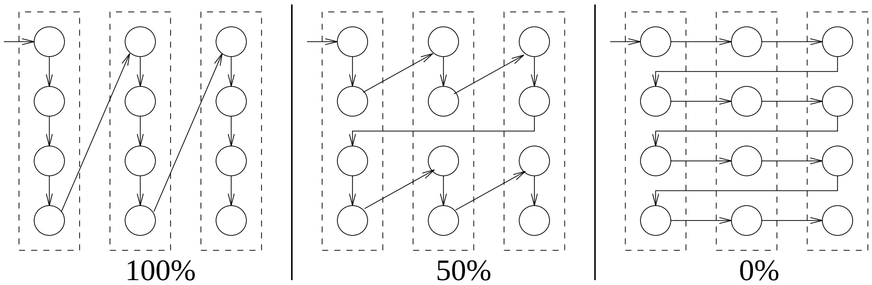

Figure 6: Clustering Factors

6.1 Experiment 1: Overhead

The first experiment measures the overhead that is incurred during normal operation with no garbage collection running. The overhead is due to extra work required to maintain the garbage collector data structures (e.g., the PRT and COT). This includes the extra log-related work that clients must perform and the processing of log records at the server. In this experiment, a single client process was run on a SPARCstation IPC with 32MB of memory; it was connected to the server over an Ethernet.

Experiments were run using four different operations: 1) object allocation, 2) modification of references in existing objects, 3) modification of non-reference data in existing objects, and 4) read-only access to objects. For each test, the operation was performed on 100,000 different objects before committing (this represents 1190 pages). Client and server cache sizes were 1500 pages — more than enough to hold all of the accessed pages. The tests were performed on the 100% clustered partition, although with the relatively large cache sizes, all clusterings would have similar performance. Each benchmark was run 10 times and results averaged. For each experiment, we report times for both a cold server cache and a hot server cache (except for allocate, which creates all of the pages it accesses).

The results of these tests are shown in Table 1. For each test, two sets of numbers are given — the times for the test without commit, and the total time for the test including commit. Commit time includes the extra garbage collection overhead of flagging PRT and/or COT entries.

As can be seen in the table, the overhead imposed on normal operations by the garbage collection code is quite small in all of the cases tested. The highest overheads were seen for the allocation of new objects; this is due to EXODUS' full-page logging for newly allocated pages. In this case, the server must scan the entire page in order to locate any cross-partition pointers. Note that individual object creations on existing pages would not incur this cost.

6.2 Experiment 2: Off-Line Garbage Collector Performance

The second experiment examines the cost of the garbage collector when it is run without any concurrent user transactions. In this case, we varied the clustering of the partition, the size of the partition, and the garbage %. All experiments were run with a server cache of 1000 pages. The partition size was varied from 500 pages (4 MB) to 10,000 pages (80 MB).

Server Cache	*Cold*			*Hot*		
Operation	**Original**	**w/GC code**	**Overhead**	**Original**	**w/GC code**	**Overhead**
Allocate	38176	40382	5.8%			
w/Commit	54989	59049	7.4%			
Update Ref	52543	53165	1.1%	34367	34920	1.6%
w/Commit	62736	63381	1.0%	44607	45160	1.2%
Update Value	51091	51616	1.0%	33104	33438	1.0%
w/Commit	61365	61998	1.0%	43327	43671	0.7%
Read-only	27586	27792	0.7%	13403	13565	1.2%
w/Commit	27622	27828	0.7%	13439	13601	1.2%

Table 1: Client Slowdown (msec), Cold and Hot Server Cache

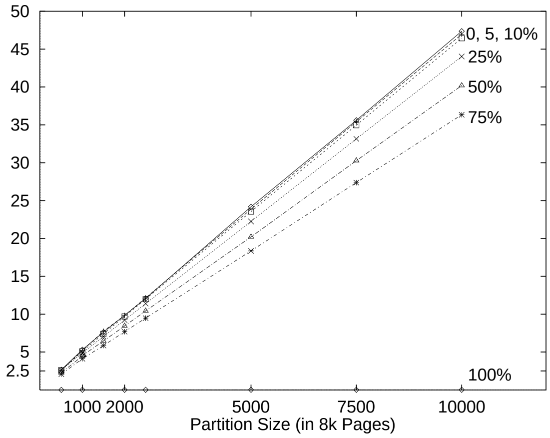

Figure 7: Marker Performance (seconds) (100% Clustering)

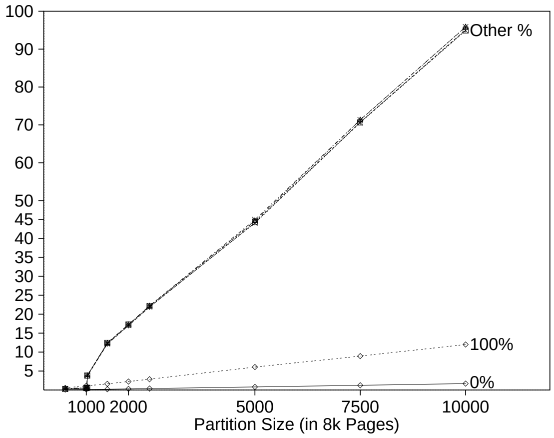

Figure 8: Sweeper Performance (seconds) (All Clusterings)

Figure 7 shows the performance of the *marking* phase of the garbage collector with 100% clustering for various percentages of garbage. In this case the marker performance scales linearly with the partition size for all % garbage values (except for 100% which remains near 0, as there is nothing for the marker to do). This linearity is because the partition size does not impact the marker, as it scans the partition sequentially in this case (due to the 100% clustering) and does the same amount of work per page regardless of the partition size. The reduction in response time as garbage % increases is because the marker traverses only live objects, and there are fewer of these.

The performance of the *sweeping* phase of the collector for the corresponding cases is shown in Figure 8. First, notice that all % garbage values have similar sweeper performance except for 0% and 100%. If the color map shows that page has 0% garbage, then the sweeper does not bother to fetch the page from disk. Likewise, if a page contains only garbage, the sweeper can free the page without fetching it from disk. The difference in performance here, comes from the overhead for freeing a page in EXODUS. In all other cases, the sweeper performance is virtually independent of the percentage of garbage in a page, as it must fetch each page that needs to be swept. When the partition is smaller than 1000 pages, all of the pages that the sweeper would need to fetch are already in the buffer because of the marker. Once the partition exceeds the cache size, then this pre-fetching effect is completely lost (in this case), as the layout of the objects in the partition cause the marker and sweeper to scan the partition in the same order.

The 100% clustering case is in some sense the best case for the garbage collector, as it allows the marking phase to process pages sequentially. Figure 9 shows the performance of the marking phase using four different clustering schemes. We show only the results for the case where the

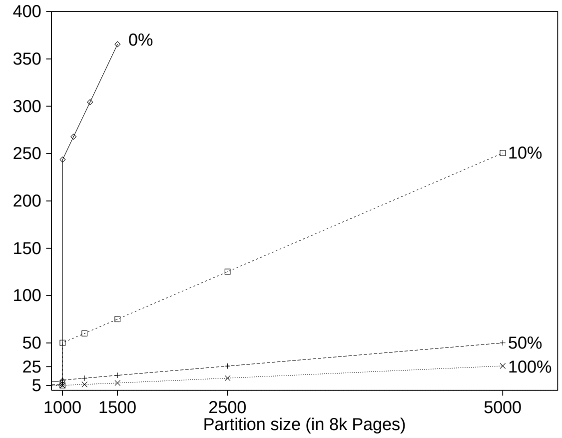

Figure 9: Marker Speed varying Locality Factor (seconds)

partition size is varied from 900 pages to 5000 pages, as performance is linear (with varying slopes) beyond that region. When the partition size is smaller than the buffer size, then the clustering factor does not affect the performance of the marker. However, once the partition exceeds the size of the buffer, then the marker begins to incur I/O due to the inter-page references. This effect is of course, to be expected and raises the issue of the garbage collector's impact on concurrent user transactions. This issue is addressed in the following section.

6.3 On-Line Performance

The third set of experiments examines the system behavior when both clients and the collector execute concurrently on the same partition. We performed measurements using two types of client programs. The first one is a read-only traversal of the partition done in a loop. The client commits after reading 10 pages. The second is a read-write traversal of the partition where a single object per page is updated. This is also done in a loop; the client commits after updating 10 pages. We vary the number of clients running in parallel from one to six. We first measure client response time while the collector is *inactive* and then do the same experiments while the collector is *active*. The response times are the average times required for each 10-page transaction. We also measure the response time of the collector itself, in order to gauge the impact of the clients on the collector.

These tests exercise the incremental aspects of the garbage collector, as the clients and the collector run concurrently. The results, therefore reflect the interaction of the `gcThread` with the

EXODUS scheduler. After numerous tests, we chose a processor time slice of 50 msec for the `gcThread` followed a 20 msec delay before it becomes eligible to run again. Furthermore, several threads for servicing client requests are allowed to run, if present, before the `gcThread` is given the processor. This scheduling technique seems to balance the collector execution and the servicing of clients. However, more study is required to better understand the interaction of the `gcThread` with the peculiarities of the EXODUS scheduler. In these experiments, the collector is run in a loop; once the partition is collected, it starts a new collection cycle. The response time for the garbage collector is the total wall clock time required for a complete collection of the partition. In these experiments we used a partition size of 1200 pages (10MB) and examined clustering factors of 100% and 50%. Each client was run on a separate SPARCstation 1+, with 32MB of memory. The client cache sizes were 200 pages and the server cache size was 1000 pages.

Figures 10 and 11 show the response time for the clients and garbage collector, respectively, for the case of 100% clustering as the number of clients is varied from one to six.

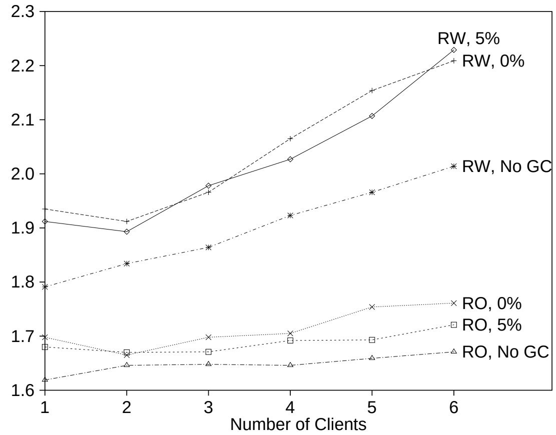

Figure 10: Client Response Time (seconds)
(100% Clustering)

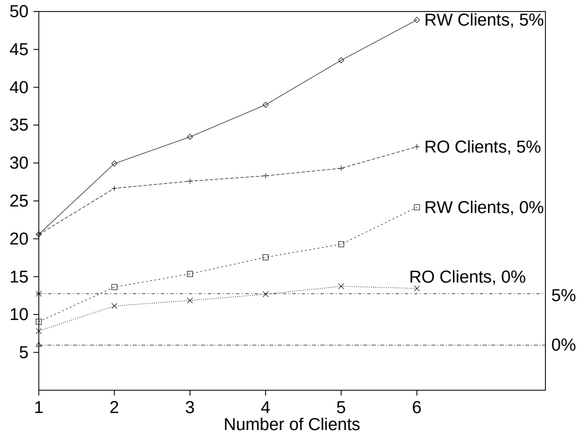

Figure 11: Collector Response Time (seconds)
(100% Clustering)

Figure 10 shows the client response times. For each experiment (read-only and read-write) three lines are shown. The lines labeled "No GC" are the times obtained by running the benchmark without running the `gcThread`. The other lines are labeled with the garbage percentage of the partition (0% or 5%). Note that the y-axis is truncated - thus, the differences among the curves here are very small. This indicates that in this case, clients are basically unaffected by the execution of the garbage collector at the server. In fact, because the collector and the clients are running on the same partition, they often pre-fetch pages for each other. Because our collector holds no locks, this pre-fetching can be fully exploited.

Figure 11 shows the response time of the garbage collector for the corresponding tests. The flat lines on the figure are the response times for the off-line collector (i.e., when no clients are active) and are used to gauge the slowdown experienced by the collector. The results show that indeed the time it takes for the collector to do its job is impacted by the amount of client activity. For example, with 6 clients running in parallel, the slowdown factor is about 2.2 for the read-only 0% garbage case and is a factor of 4 for read-write clients with 0% garbage. The slowdown for 5%

garbage is similar.

For read-write queries, the collector's response time is doubled when garbage is introduced (i.e., 5% vs. 0% garbage) This is because with garbage present in each page, the sweeping phase must refetch every page in order to sweep it. We anticipated a lower overhead in this case, because we expected that active transactions would be bringing pages into the server buffer would later be needed by the sweeper. Detailed traces of the collector's activity however, show that the sweeper needs to refetch nearly all of the partition's pages from disk. This very low cache hit ratio is due to the way the marker and the sweeper process pages: they both access the pages in the same order. This means the last pages of the database are in the cache at the beginning of the sweeping phase. However, the sweeper starts by processing the first pages of the database.

In general,these results show that the garbage collector can indeed run in a way that does not negatively impact the performance of the clients. This is again, due to the low overhead of the log-based "write barrier", the collector's tolerance of client caching and steal buffer management and because the collector does not synchronize with clients. The results of this experiment, while encouraging, point out that we need to better understand the interaction between the `gcThread` and EXODUS' scheduling of thread execution at the server. For example, in Figure 10, it can be seen that when the `gcThread` is active, the response time of a client running alone is slightly higher than when two clients are running (this is not the case if there is no garbage collection). We believe that this phenomenon is related to the way that the `gcThread` is scheduled in the presence of server idle time, however, this issue requires further investigation.

In addition to the 100% clustering case described above, we also performed the same test with a clustering factor of 50% to ensure that the low impact of the collector was not due to the presence of perfect clustering. The results of this test are summarized (for six clients) Table 2. For comparison purposes, Table 2 includes the corresponding results from the 100% clustering case described previously. Recall with a locality of 50%, the marker processes half of the page before moving on to the next page. As can be seen, the overheads on both client processing and garbage collection are similar; the overhead on clients is at most 11% for these tests and the collector takes several times longer than it would if no clients were running.

	Client Performance				*Collector Performance*			
Query type	RO		RW		RO		RW	
Locality	100%	50%	100%	50%	100%	50%	100%	50%
Alone (msec)	(1671) 100	(1912) 100	(2014) 100	(2630) 100	(5579) 100	(13096) 100	(12747) 100	(24957) 100
0% Garbage	105.3	110.8	109.6	107.1	240.9	246.2	189.5	250.5
5% Garbage	102.9	106.7	110.6	106.9	576.4	405.2	383.5	317.5

Table 2: Relative Performance varying Locality Factor

7 Related Work

As stated in the introduction, garbage collection has been intensively studied in the context of traditional programming languages. Surveys of this work include [Coh81] and [Wil92]. The invariants that an incremental collector must respect were first proposed by [DLM+78, Bak78]. Our work addresses the efficient implementation of similar invariants in a (transactional) client-server DBMS context. The earliest study of garbage collection for object-oriented databases was done by Butler [But87]. This work simulated the behavior of several kinds of collectors running against a centralized OODBMS, but did not consider interactions with concurrency control, recovery, and caching mechanisms. More recent work has investigated fault-tolerant garbage collection techniques for transactional persistent systems in centralized [KLW89, KW93, ONG93] and distributed [MRV91, MS91] architectures. This work addresses fault tolerance but does not consider dynamic page replication and caching as arises in a workstation/server environment.

A reference counting collection scheme for MIT's Thor system is described in [ML94]. Thor [LDS92] is a distributed OODBMS which uses optimistic concurrency control to regulate accesses to objects. This paper focuses on distributed collection across servers in a client-server environment rather than on collection that is local to a server. Each time a client fetches an object from a server, the server records the OID of the object into a local table. The server also records the OIDs that are referenced by the fetched object. This is to cope with the optimistic concurrency control, since fetched objects are not protected by any lock. Server tables are cleaned as a side effect of local collections. These tables can be viewed as a before image log, which avoids the reclamation of pruned objects prior to transaction commit. The algorithm uses a "no-steal" policy so that modified objects are not sent to the servers prior to commit. This policy avoids the problems due to partial flushes of updates (Section 2.3) at the expense of reduced flexibility in client cache management. [ML94] describes the algorithm but does not discuss an implementation and provides no performance analysis.

The work that is most relevant to our algorithm is [YNY94]. This paper investigates the performance tradeoffs of several reclamation algorithms for client-server persistent object stores. Some of the results are obtained from an implementation of an incremental partitioned Mark & Sweep algorithm, although very few details of this algorithm or its implementation are given. Most of the results are obtained using a simulation of several algorithms.

Similarly to our algorithm, their partitioned Mark & Sweep collector runs at the server and can execute concurrently with client transactions. However, their algorithm differs in the that it uses a special *write barrier* and that the collector obtains (non-two phase) locks on data items. The write barrier traps updates at the clients and adds any new object references to a local list. This list is shipped to the server when a client commits or when the client receives a callback message from the server. The server requests these lists and the contents of application process stacks from the clients before the collector enters its sweep phase. The lists and other references are used during the marking phase as additional roots of persistence. In terms of locks, the marking phase obtains and holds a read lock on a page while it is accessing the page. These locks cause the marker to synchronize with the transactions. In contrast, our partitioned Mark & Sweep algorithm does not hold any locks on pages, does not send callbacks to clients, and can ignore the program state of on-going transactions. Measurements of the implementation showed that the write barrier has only minimal impact on client performance; our measurements support this result.

Several other algorithms are examined in [YNY94], including a partitioned copy-based collection algorithm. This algorithm obtains non-two-phase exclusive transactional locks for moving objects and uses callbacks, it also requires the use of logical OIDs. Based on the results of the simulation studies, the copy-based algorithm is advocated over partitioned Mark & Sweep due to its ability to recluster the database. In contrast, we have chosen to allow clustering to be treated separately by the system in order to gain the efficiency and relative ease of implementation of Mark & Sweep.

8 Conclusions

In this paper, we first described the requirements imposed on a garbage collector suited for workstation/server OODBMS environments. The need for efficient garbage collection in the presence of client caching, "steal" management of client buffers, transactions, and fault tolerance raise three major problems that need require special care. These problems are tied to the rollback of transactions, the partial flushing of multi-page updates involving object creations and reference updates, and the potential for overwriting of garbage collected pages due to recovery and/or client caching.

We described a garbage collection algorithm based on a partitioned Mark & Sweep approach. By exploiting the flow of log records between clients and server, we are able to enforce the correctness of our algorithm. The collector is incremental, but requires very little synchronization with client transactions (e.g., it holds no locks), performs minimal logging, and requires no client callbacks or special hardware. The algorithm has been implemented in an existing OODBMS (i.e., EXODUS), and integrated with the concurrency control and recovery of that system. Furthermore, the garbage collector affected fewer than 300 lines of code in the client side of the system. An initial study showed the performance characteristics of our implementation and suggests that the impact of the collector on the activity of clients is minimal.

We are currently implementing the mechanisms to allow the collector to handle cross-partition references. As stated, previously, most of the bookkeeping features needed for this extension are already implemented and included in the measurements in the performance study. Furthermore, we are more closely investigating the interaction between the garbage collector thread and the EXODUS thread scheduling mechanism at the server. In terms of future work, we plan to use our implementation to investigate data partitioning and partition selection strategies, and to more closely explore alternative approaches such as scavenging.

Acknowledgements

We would like to thank Mike Zwilling, who provided invaluable assistance and information about EXODUS on many occasions during this work.

References

[Bak78] H. Baker. List Processing in Real Time on a Serial Computer. *Com. of the ACM*, 21(4):280–294, April 1978.

[BD90] V. Benzaken and C. Delobel. Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2. In *4th Int. Workshop on Persistent Object Systems*, pages 403–412, Martha-Vineyard, Mass., September 1990.

[BDK91] F. Bancilhon, C. Delobel, and P. Kannellakis. *Building an object-oriented database : the O_2 story.* Morgan Kaufmann, 1991.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Com. of the ACM*, 34(10), October 1991.

[But87] M. Butler. Storage Reclamation in Object Oriented Database Systems. In *Proc. of the ACM SIGMOD Int. Conf.*, pages 410–425, San Francisco, CA, May 1987.

[Cat94] R. Cattell. *The ODMG Object Database Standard, Rel 1.1.* Morgan-Kaufman, San Mateo, CA, 1994.

[CDN93] M. Carey, D. Dewitt, and J. Naughton. The OO7 Benchmark. In *Proc. of the ACM SIGMOD Int. Conf*, Washington D.C., May 93.

[CDRS86] M. Carey, D. DeWitt, J. Richarson, and E. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proc. of the 12th VLDB Int. Conf.*, Kyoto, Japan, 1986.

[CFLS91] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proc. of the ACM SIGMOD Int. Conf.*, Denver, June 1991.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proc. of the ACM SIGMOD Int. Conf.*, Minneapolis, MN, May 94.

[Coh81] J. Cohen. Garbage Collection of Linked Data Structures. *Computing Surveys*, 13(3):341–367, September 1981.

[Col60] G. Collins. A method for overlapping and erasure of lists. *Com. of the ACM*, 2(12):655–657, December 1960.

[CWZ94] J. Cook, A. Wolf, and B. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *Proc. of the ACM SIGMOD Int. Conf.*, pages 371–382, Mineapolis, MN, May 1994.

[DB76] L. Deutsch and D. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *Com. of the ACM*, 19(9):522–526, September 1976.

[DFMV90] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Proc. of the 16th VLDB Int. Conf.*, Brisbane, Australia, August 1990.

[DLM+78] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Com. of the ACM*, 21(11):966–975, November 1978.

[Exo93] EXODUS Project Group. EXODUS Storage Manager Architectural Overview, 1993.

[FC92] M. Franklin and M. Carey. Client-Server Caching Revisited. In *Proc. of the Int. Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[FCL93] M. Franklin, M. Carey, and M. Livny. Local Disk Caching in Client-Server Database Systems. In *Proc. of the 19th VLDB Int. Conf*, Dublin, Ireland, August 1993.

[FCW89] M. Franklin, G. Copeland, and G. Weikum. What's Different About Garbage Collection For Persistent Programming Languages? Technical Report ACA-ST-062-89, MCC, Austin, Texas, February 1989.

[FZT+92] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proc. of the ACM SIGMOD Int. Conf.*, San Diego, June 1992.

[GA92] O. Gruber and L. Amsaleg. Object Grouping in Eos. In *Proc. of the Int. Workshop on Distributed Object Management*, pages 117–131, Edmonton, Canada, August 1992.

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

[Hug85] J. Hughes. A Distributed Garbage Collection Algorithm. In *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[KLW89] E. Kolodner, B. Liskov, and W. Weihl. Atomic Garbage collection: Managing a Stable Heap. In *Proc. of the ACM SIGMOD Int. Conf.*, pages 15–25, Portland, Oregon, June 1989.

[KW93] E. Kolodner and W. Weihl. Atomic incremental garbage collection and recovery for large stable heap. In *Proc. of the ACM SIGMOD Int. Conf.*, Washington D.C., June 1993.

[LDS92] B. Liskov, M. Day, and L. Shrira. Distributed Object Management in Thor. In *Proc. of the Int. Workshop on Distributed Object Management*, pages 79–91, Edmonton, Canada, August 1992.

[McB63] J. McBeth. On the Reference Counter Method. *Com. of the ACM*, 6(9):575, September 1963.

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Com. of the ACM*, 3(4):184–195, April 1960.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), March 1992.

[ML94] U. Maheshwari and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database. In *Proc. of the 3rd PDIS Int. Conf.*, Austin, TX, September 1994.

[MN94] C. Mohan and I. Narang. ARIES/CSA: A method for Database Recovery in Client-Server Architectures. In *Proc. of the ACM SIGMOD Int. Conf.*, Minneapolis, MN, May 1994.

[MRV91] L. Mancini, V. Rotella, and S. Venosa. Copying Garbage Collection for Distributed Object Stores. In *10th Symposium on Reliable Distributed Systems*, Pisa, Italy, September 1991.

[MS91] L. Mancini and S. Shrinivastava. Fault-tolerant Reference Counting for Garbage Colection in Distributed Systems. *Computer Journal*, 34(6):503–513, December 1991.

[ONG93] J. O'Toole, S. Nettles, and D. Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proc. of the 14th SOSP*, pages 161–174, Asheville, NC, December 1993. ACM Press.

[SGP90] M. Shapiro, O. Gruber, and D. Plainfossé. A Garbage Detection Protocol for a Realistic Distributed Object-support System. Technical Report INRIA-1320, INRIA, Rocquencourt, November 1990.

[Wil92] P. Wilson. Uniprocessor Garbage Collection Techniques. In *Int. Workshop on Memory Management*, volume 637, pages 1–43, St. Malo, France, September 1992. Springer-Verlag.

[YNY94] V. Yong, J. Naughton, and J. Yu. Storage Reclamation and Reorganization in Client-Server Persistent Object Stores. In *Proc. of the Data Engineering Int. Conf.*, pages 120–133, Houston, TX, February 1994.

[ZM90] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, CA, 1990.