Garbage Collection in Object Oriented Databases Using Transactional Cyclic Reference Counting

S. Ashwin[1*] Prasan Roy[1] S. Seshadri[1] Avi Silberschatz[2] S. Sudarshan[1]

[1]Indian Institute of Technology,
Mumbai 400 076, India
sashwin@cs.wisc.edu
{prasan,seshadri,sudarsha}@cse.iitb.ernet.in

[2]Bell Laboratories
Murray Hill, NJ 07974
avi@bell-labs.com

Abstract

Garbage collection is important in object-oriented databases to free the programmer from explicitly deallocating memory. In this paper, we present a garbage collection algorithm, called Transactional Cyclic Reference Counting (TCRC), for object oriented databases. The algorithm is based on a variant of a reference counting algorithm proposed for functional programming languages The algorithm keeps track of auxiliary reference count information to detect and collect cyclic garbage. The algorithm works correctly in the presence of concurrently running transactions, and system failures. It does not obtain any long term locks, thereby minimizing interference with transaction processing. It uses recovery subsystem logs to detect pointer updates; thus, existing code need not be rewritten. Finally, it exploits schema information, if available, to reduce costs. We have implemented the TCRC algorithm and present results of a performance study of the implementation.

* Currently at the University of Wisconsin, Madison

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

1 Introduction

Object oriented databases (OODBs), unlike relational databases, support the notion of object identity, and objects can refer to other objects via object identifiers. Requiring the programmer to write code to track objects and their references, and to delete objects that are no longer referenced, is error prone and leads to common programming errors such as memory leaks (garbage objects that are not referred to from anywhere, and haven't been deleted) and dangling references. While these problems are present in traditional programming languages, the effect of a memory leak is limited to individual runs of programs, since all garbage is implicitly collected when the program terminates. The problem becomes more serious in persistent object stores, since objects outlive the programs that create and access them. Automated garbage collection is essential in an object oriented database to protect from the errors mentioned above. In fact, the Smalltalk binding for the ODMG object database standard requires automated garbage collection.

We model an OODB in the standard way as an *object graph*, wherein the nodes are the objects and the arcs are the references between objects. The graph has a persistent *root*. All objects that are reachable from the persistent root or from the transient program state of an on-going transaction are *live*; while the rest are *garbage*. We often call object references as *pointers*.

There have been two approaches to garbage collection in object oriented databases: *Copying Collector* based [YNY94] and *Mark and Sweep* based [AFG95]. The copying collector algorithm traverses the entire object graph and copies live objects into a new space; the entire old space is then reclaimed. In contrast, the Mark and Sweep algorithm marks all live objects by traversing the object graph and then traverses (sweeps) the entire database and deletes all objects that are unmarked. The copying collector algorithm reclusters objects dynamically; the reclustering can improve locality of reference in some cases, but may destroy programmer specified clustering resulting in worse performance in other cases. The garbage collection algorithms of [YNY94] as well as [AFG95] handle concurrency control and recovery issues.

With both the above algorithms, the cost of traversing the entire object graph can be prohibitively expensive for databases larger than the memory size, particularly if there are many cross-page references. In the worst case, when the buffer size is a small fraction of the database size and objects in a page refer to objects in other pages only, there may be an I/O for every pointer in the database. To alleviate this problem, earlier work [YNY94, AFG95] has attempted to divide the database into *partitions* consisting of a few pages. Each partition stores inter-partition references, that is references to objects in the partition from objects in other partitions, in a persistent data structure. Objects referred to from other partitions are treated as if they are reachable from the persistent root, and are not garbage collected even if they are not referred to from within the partition. Each partition is garbage collected independent of other partitions; references to objects in other partitions are not followed. Thus, partitioning makes the traversal more efficient; the smaller the partition, the more efficient the traversal, with maximum efficiency occurring if the whole partition fits into the buffer space.

Unfortunately, small partitions increase the probability of self-referential cycles of garbage that cross partition boundaries; such cyclic garbage is not detected by the partitioned garbage collection algorithms. Previous work has maintained that such cross cycle structures will be few, and will "probably" not be a problem. However, simulations by [CWZ94] showed that even small increases in database connectivity can produce significant amounts of such garbage. Therefore, it is not clear that partition sizes can be made very small without either failing to collect large amounts of garbage or employing special (and expensive) techniques to detect such cyclic garbage.

A natural alternative is *Reference Counting*. Reference Counting is based on the idea of keeping a count of the number of pointers pointing to each object. When the reference count of the object becomes zero, it is garbage and eligible for collection. Reference counting has the attractive properties of localized and incremental processing. Unfortunately, basic reference counting cannot deal with self-referential cycles of objects; each object could have a positive reference count, yet all the objects in the cycle may be unreachable from the persistent root, and therefore be garbage. However, a number of extensions of the basic referencing counting algorithm to handle cyclic data have been proposed in the programming language community, including: [Bro85, Bro84, PvEP88]. More recent work in this area includes [Lin90, MWL90, JL91].

In this paper, we consider a version of reference counting, proposed by Brownbridge [Bro85, Bro84] for functional programming languages, which handles self referential cycles of garbage. We present an algorithm, called Transactional Cyclic Reference Counting (TCRC), based on Brownbridge's algorithm, which is suitable for garbage collection in an OODB. The salient features of the TCRC algorithm are:

- It detects all self referential cycles of garbage unlike basic reference counting, and the partitioned garbage collection algorithms.
- It performs a very localized version of mark-and-sweep to handle cyclic data, with each mark-and-sweep likely to access far fewer objects than a global mark-and-sweep. Thus it does not have to examine the entire database while collecting garbage, except in the worst case.
- It allows transactions to run concurrently, and does not obtain any long term locks, thereby minimizing interference with transaction processing.
- It is integrated with recovery algorithms, and works correctly in spite of system crashes. It also uses recovery subsystem logs to detect pointer updates; thus, existing application code need not be rewritten.
- It exploits schema information, if available, to reduce costs. In particular, if the schema graph is acyclic, no cyclic references are possible in the database and TCRC behaves identically to reference counting.

A proof of correctness of the TCRC algorithm is presented in [ARS+97]. Designing a cyclic referencing counting algorithm which allows concurrent updates and handles system crashes is rather non-trivial, and to our knowledge has not been done before; we believe this is one of the central contributions of our paper.

A problem often cited against reference counting schemes is the overhead of updating reference counts. However, each pointer update can only result in at most one reference count being updated. This overhead will have only a small impact on performance if, as we expect is true in any realistic scenario, pointer updates are only a small fraction of the overall updates. For TCRC, moreover, the overhead is offset by the reduced cost of traversals while collecting garbage.

We have implemented a prototype of the TCRC algorithm as well as the partitioned mark and sweep algorithm on a storage manager called *Brahmā* developed in IIT Bombay. We present a performance study of TCRC based on the implementation; the study clearly illustrates the benefits of TCRC.

2 Brownbridge's Cyclic Reference Counting Algorithm

Our Transactional Cyclic Reference Counting algorithm is based on the Cyclic Reference Counting (CRC) algorithm proposed by Brownbridge [Bro84, Bro85], in the context of functional programming languages.

The basic idea behind the Cyclic Reference Counting (CRC) algorithm of Brownbridge [Bro84, Bro85] is to label edges in the object graph as *strong* or *weak*. The labelling is done such that a cycle in the object graph cannot consist of strong edges alone – it must have at least one weak edge. Two separate reference counts for strong and for weak edges (denoted SRefC and WRefC respectively) are maintained per object. It is not possible in general to cheaply determine whether labelling a new edge as strong creates a cycle of strong edges or not. Hence, in the absence of further information, the algorithm takes the conservative view that labelling a new edge strong could create a cycle of strong edges, and labels the new edge weak.

The SRefC and WRefC are updated as edges are created and deleted. If for an object S, the SrefC as well as WrefC is zero, then S is garbage and S and the edges from it are deleted. If the SrefC is zero, but WrefC is non-zero, there is a chance that S is involved in a self referential cycle of garbage. If the SrefC of an object S is greater than zero, then S is guaranteed to be reachable from the root (however, our TCRC algorithm does not guarantee this last property).

If the object graph did not have any garbage before the deletion of an edge to S, then the only potential candidates for becoming garbage are S and objects reachable from S. If SrefC of S is zero and WrefC of S is nonzero, a localized mark and sweep algorithm detects whether S and any of the objects reachable from S are indeed garbage. The localized mark and sweep performs a traversal from S and identifies all objects reachable from S and colours them red. Let us denote the above set by R. It then colours green every object in R that has a reference from an object outside R (detected using reference counts). It also colours green all objects reachable from any green object. During this green marking phase some pointer strengths are updated to ensure that every object has at least one strong pointer to it. We will describe this pointer strength update in detail in the context of our transactional cyclic reference counting algorithm. At the end, all objects in R not marked green are garbage and are deleted.

However, prior cyclic reference counting algorithms, including Brownbridge's algorithm, were designed for a single user system. They cannot be used in a multi-user environment with concurrent updates to objects, and do not deal with persistent data and failures. Our contributions lie in extending Brownbridge's algorithm to (a) use logs of updates to detect changes to object references, (b) to work in an environment with concurrent updates, (c) to work on persistent data in the presence of system failures and transaction aborts, (d) handle a batch of updates at a time rather than one update at a time, and (e) optimize the localized mark and sweep significantly by following only strong pointers.

3 System Model and Assumptions

In this section, we describe our system model and outline the architectural assumptions on which our garbage collector is based, which is very similar to the model and assumptions in [AFG95].

In our model, transactions log undo and redo information for all updates. Undo and redo records are represented as undo(tid, oid, offset, old-value), and redo(tid, oid, offset, new-value), where tid denotes a transaction identifier and oid an object identifier. Object creation is logged as object-allocation(tid, oid). The commit log is represented as commit(tid); and the abort log is represented as abort(tid). We require that from the oid we can identify the type of the object (perhaps by first fetching the object), and from the offset we can determine if the value that has been updated is a pointer field. These requirements are satisfied by most database systems.

We make the following important assumption about transactions:

Assumption 3.1 *Transactions follow strict two-phase locking on objects. That is, transactions acquire read or write locks on objects as appropriate, and hold read as well as write locks until end of transaction.* □

As with any other garbage collection scheme, we assume that an object identifier is valid only if it is either a persistent root, or is present in a pointer field of an object in the database, or is in the transient memory (program variables or registers) of an active transaction that read the value from an object in the database. Note that this precludes transactions from passing oids to other transactions, and from storing oids in external persistent storage.

Our algorithms can be used in centralized as well as client-server settings. Let us consider first the centralized setting.

Assumption 3.2 *In the centralized setting we assume that transactions follow* **strict WAL**, *that is,* *they log both the undo and the redo value before actually performing the update.* □

Our algorithms also work in a data-shipping client-server environment, under the following assumptions

Assumption 3.3 *In the client-server setting we assume that clients follow:*

1. **strict WAL** *with respect to the server. That is, before any data is received by the server, the undo as well as redo information for the data must have already been received by the server.*

2. **force** *with respect to the server. That is, before the transaction commits, all the updated data must have been received by the server.* □

These assumptions make the client transaction behave, as far as the server is concerned, just like a local transaction that follows strict WAL.

Our techniques are not affected by the unit of data shipping (such as page or object) and whether or not data is cached at the client. The clients can retain copies of updated data after it has been sent to the server.

Most of the assumptions above are satisfied by typical storage managers for object-oriented databases. Our client server assumptions are also very similar to those of [AFG95].

4 Transactional Cyclic Reference Counting

We will now describe the Transactional Cyclic Reference Counting (TCRC) algorithm. We first describe the data structures needed by the transactional cyclic reference counting algorithm.

4.1 Data Structures

Associated with each object, we persistently maintain a strong reference count (SRefC) giving the number of strong pointers pointing to the object, a weak reference count (WRefC) giving the number of weak pointers pointing to the object, and a strength bit for the object. Each pointer also has a strength bit. Both strength bits are persistent. The pointer is strong if the strength bit in the pointer and the strength bit in the object pointed to have the same value; otherwise the pointer is weak. This representation of strength using two bits is an important implementation trick, from Brownbridge [Bro85, Bro84]. It makes very efficient the operation of *flipping the strength* of all pointers to an object, that is making all strong pointers to the object weak, and all weak pointers to the object strong. All that need be done is to flip the value of the strength bit in the object.

The TCRC algorithm also maintains a persistent table, the *Weak Reference Table* (WRT), which contains oids for the objects which have a zero SRefC, i.e. no strong pointers incident on them. The persistent root is never put into the WRT.

All the above information can be constructed from the object graph and therefore it could be made transient. However, we would then have to reconstruct the information after a system crash by scanning the entire database, which would be expensive. Hence we make it persistent. Updates to SRefC and WRefC, update of the strength bit of an object or of a pointer, and the insert or delete of entries from the WRT are logged as part of the transaction whose pointer update caused the information to be updated/inserted/deleted.

There is also a non-persistent table which is used during garbage collection: the *Red Reference Table* (RRT); this table associates with (some) objects a *strong red reference count* (SRedRefC), a *weak red reference count* (WRedRefC), and a bit that indicates whether the colour of the object is red or green. This table is stored on disk since the size of this table could be large in the worst case, but updates to this table are not logged.

Finally, similar to [AFG95] TCRC maintains an non-persistent in-memory table called the *Temporary Reference Table* (TRT), which contains all those oids such that a reference to the object was added or deleted by an active transaction, or the object was created by the transaction. Such an oid may be stored in the transient memory of an active transaction although the object may not be referenced by any other object in the database. An object whose oid is in TRT may not be garbage even if it is unreachable from any other object, since the transaction may store a reference to the object back in the database. Updates to TRT are also not logged. The TRT also provides a simple way of handling the persistent root — its oid is entered in the TRT at system start up, and is never removed. This prevents the garbage collector from collecting the persistent root.

4.2 The Algorithm

TCRC consists of two distinct algorithms, run by different processes. The first is the *log-analyzer* algorithm. The second algorithm is the actual *garbage collection* algorithm. We describe them below.

4.2.1 Log analyzer

The log-analyzer algorithm analyzes log records generated by the transaction, and performs various actions based on the log records. As part of its actions, it may also insert records into the log. We shall assume it is run as part of the transaction itself, is invoked each time a log record is appended to the system log tail, and is atomic with respect to the appending of the log record.

In the actual implementation, it is possible to run the log-analyzer as a separate thread, and when a transaction appends a log record to the system log, it actually only delivers it to the log-analyzer, which then appends the log record to the system log. In particular, in the client-server implementation the log-analyzer process is run at the server end, not at the client.

The log-analyzer makes use of the following procedures. Procedure DeletePointer decrements the WRefC or SRefC for an object when a pointer to the object is deleted. If the SRefC falls to zero after the decrement then the object's oid is put into WRT. Procedure AddPointer, by default, sets the strength of the pointer to be weak and increments the WRefC of the object pointed to. The strength is set to weak so that cycles of strong edges are not created; however, we will see in Section 5 that we may be able to make some new pointers strong.

The procedure LogAnalyzer works as follows. First it obtains the log_analyzer_latch (which is also acquired by the garbage collection thread) to establish a consistent point in the log. The latch is obtained for the duration of the procedure. The log is analyzed by the log analyzer and depending on the type of the log record various actions as outlined below are taken. For undo/redo log records caused by pointer updates, the reference counts for the affected objects are updated. This is done by DeletePointer in case of undo logs, and AddPointer in case of redo logs. For log records corresponding to the allocation of objects, the reference counts for the new object are initialized to zero, and the oid of the object is inserted into the WRT. In all the above cases (i.e., for pointer updates and object allocation), the oid of the affected object is inserted into the TRT with the tid of the transaction that generated the record.

For end-of-transaction (commit or abort) log records, the algorithm first tries to get the gcLatch. If the latch is obtained immediately, then garbage collection is not in progress and all the oid entries for the terminating transaction from the TRT are removed and the gcLatch released thereafter. However, if the gcLatch cannot be obtained immediately then a garbage collection is in progress concurrently. In this case, the oid entries for the terminating transaction are not removed, but instead flagged for later removal by the garbage collector.

All operations on pointer strengths and reference counts are protected by a latch on the object pointed to, although not explicitly mentioned in our algorithms. Access to WRT and TRT are also protected by latches.

```
Procedure CollectGarbage {
    acquire gcLatch
    RRT = {}
S1: for each oid in WRT that is not in TRT
        RedTraverse(oid)
S2: for each oid ∈ RRT
        latch the reference count entry of oid
        if SRefC_oid + WRefC_oid >
                SRedRefC_oid + WRedRefC_oid
            mark oid as green;
        unlatch reference count entry of oid
    for each oid ∈ RRT that is marked green
        if SRefC_oid == SRedRefC_oid
            /* all external pointers
                to the object are weak */
            if SRefC_oid == 0 /* oid is in WRT */
                remove oid from WRT
            flip the strength of all pointers to oid
            swap SRefC_oid and WRefC_oid
        GreenTraverse(oid)
    done = FALSE
S3: while done == FALSE
        done = TRUE
        acquire log_analyzer_latch
S4:     for each oid ∈ RRT that is marked red
            if oid ∈ TRT
                release log_analyzer_latch
                GreenTraverse(oid)
                done = FALSE
                acquire log_analyzer_latch
        release log_analyzer_latch
S5: for each oid ∈ RRT that is marked red
        Collect(oid)
    release gcLatch
    remove all flagged entries from TRT
}

Procedure GreenTraverse(oid) {
    starting with oid as the root do a
        depth-first traversal restricted to
        the objects marked red in RRT
    when visiting an object during the traversal :
        mark the object green
        make strong all pointers from the object
            to any red object (not yet visited)
        make weak all pointer from the object to
            any green object (already visited)
}
```

Figure 1: Pseudo Code for Garbage Collector

4.2.2 Garbage Collector

The garbage collection algorithm is activated periodically (possibly depending on availability of free space). The algorithm makes use of the following support functions. Procedure Collect actually deletes an object; before doing so, it deletes all pointers out of the object, updating the stored reference counts of the objects pointed to. It also deletes the object from RRT and WRT.

Procedure RedTraverse performs a reachability scan from the specified object, following only strong pointers, and marks all reachable objects red and puts them in RRT. RedTraverse also maintains for each object present in RRT, two counts: $SRedRefC$ and $WRedRefC$, giving respectively the number of strong and weak pointers to the object from all other objects present in RRT. These counts are maintained on the fly during the traversal; in order to do so, RedTraverse also maintains these counts for objects that are reachable by a single weak edge from objects in RRT, since such objects may be added to RRT later in the scan.

The garbage collection algorithm is implemented by Procedure CollectGarbage, shown in Figure 1. Initially, all nodes reachable from objects in WRT using only the strong pointers are coloured red and put in RRT by calling RedTraverse. This function performs a fuzzy localized traversal of the object graph during which no locks are obtained on the objects being traversed. Short term latches may be obtained on objects or pages to ensure physical consistency.

After this, in Step S2 some nodes are marked green based on the values of their WRefC+SRefC and WRedRefC+SRedRefC. WRedRefC is the number of weak pointers pointing to an object amongst pointers from objects in RRT. Similarly, SRedRefC is the number of strong pointers pointing to an object amongst pointers from objects in RRT. The expression WRedRefC + SRedRefC counts how many pointers to a node s are from nodes in RRT. If this count is less than the total number of pointers to node s, there must be an external (to objects in RRT) pointer to s, and s is not garbage. Such objects are marked green in Step S2. The Procedure GreenTraverse called in Procedure CollectGarbage can be found in Figure 1.

Next, in Step S4, any objects in RRT that are in TRT are also marked green since their references may still be stored in an ongoing transaction and stored back in the database. Objects that are reachable from the above objects are also marked green, by invoking GreenTraverse. The reason for performing Step S4 repeatedly (in the while loop at Step S3) is to establish a consistent point in the log at which no object in the RRT is in TRT; this helps simplify the proof of correctness. Let us denote the time instant when we acquire the log_analyzer_latch for the last time in the while loop at step S3 as T5. This guarantees that all objects in RRT that are marked red at step S5 are not in TRT according to log at T5.

4.2.3 Support for Logical Undo by the Recovery Manager

The TCRC algorithm needs some support from the recovery manger in the form of supporting logical undos to ensure correctness. There are some actions whose undos have to be performed logically and not physically. We discuss them below and discuss what the logical undo should do in each case:

Pointer Deletion and Strength Update: Undo of a pointer deletion or strength update, if performed naively, may introduce strong cycles in the graph, which can affect the correctness of the algorithm. The right way to undo a pointer deletion is to reinsert the pointer with the strength set to be weak (even if it was strong earlier). Similarly, the undo of a pointer strength update (done in case of system crash during the garbage collection phase) is to set the strength of the pointer as weak (irrespective of the original strength).

Reference Counts Update: The reference counts of an object O can be concurrently updated by multiple transactions (including the garbage collector) through different objects which are locked by the transactions. The object O itself need not be locked since only a reference to it is being updated. Only short term latches are necessary for maintaining physical consistency. If a transaction that updated the reference count of an object aborts, it should be logically undone: the undo of a reference count increment is a decrement of the same reference count, while the undo of a reference count decrement is always an increment of WRefC since a reinserted pointer is always weak.

4.3 Correctness

Theorem 4.1 *The TCRC algorithm*

1. *eventually collects any object that is garbage.*
2. *does not incorrectly reclaim live objects as garbage.* □

The above theorem establishes the correctness of the TCRC algorithm; a proof is presented in [ARS+97]. The theorem holds in the presence of concurrent transactions and system failures.

An interesting point to note is that RedTraverse follows only strong pointers, and not weak pointers, in contrast to Mark-and-Sweep. Our proof of correctness shows that every garbage object is either in WRT or is reachable by a sequence of strong edges from an object in WRT, and thus RedTraverse finds all garbage objects. We also show that all non-garbage objects coloured red are later coloured green by a call on GreenTraverse, even though GreenTraverse only follows edges through red objects.

Another interesting point is that although our traversals (both RedTraverse and GreenTraverse) are fuzzy, that is they do not acquire any long term locks, the algorithms are still correct. The TRT (also used by [AFG95]) plays an important role here, since any pointers that are added or deleted during the traversal are inserted into the TRT. Objects reachable from TRT are not garbage collected.

A badly designed garbage collection algorithm could create infinite work for itself, by leaving oids in WRT which will be traversed by another garbage collection phase, which in turn leaves oids in WRT, ad infinitum. We now state a theorem which guarantees that this does not happen; that is, in the absence of updates, the system eventually reaches a state where garbage collection thread does no more work.

Theorem 4.2 *If there are no updates from the beginning of one garbage collection phase to the end of the next garbage collection phase no object will be in WRT at the end of the second garbage collection phase.* □

The proof is presented in [ARS+97].

5 Using the Schema Graph

We now see how to use information from the database schema to optimize TCRC. The schema graph is a directed graph in which the the nodes are the classes in the schema. An edge from node i to node j in the schema graph denotes that Class i has an attribute that is a reference to Class j. The pointers in the schema graph thus form a template for the pointers between the actual instances of the objects. If an edge E in a schema graph is not involved in a cycle, then neither can an edge e in the object graph for which E is the template.

We label edges which are not part of a cycle in the schema graph as *acyclic* and the others as *cyclic*. When adding an edge e to the object graph, if its corresponding template edge in the schema graph is acyclic, the strength of e is set to be *strong*. During garbage collection, in RedTraverse, we do not follow strong edges whose template edge is acyclic. In the extreme case where the schema graph is acyclic, no edges are traversed, and TCRC behaves just like reference counting, reducing the cost significantly.

6 Performance Evaluation

We implemented the TCRC algorithm and the Partitioned Mark and Sweep (PMS) algorithm on an object manager called *Brahmā* developed at IIT Bombay. *Brahmā* supports concurrent transactions using two phase locking and a complete implementation of the ARIES recovery algorithm. It provides extendible hash indices as well as B^+-tree indices as additional access mechanisms.

The WRT is implemented as a persistent extendible hash table indexed on the oid while the TRT is an in-memory hash table indexed separately on the oid and the transaction id (to allow easy deletion of all entries of a transaction). The reference counts SRefC and WRefC are stored with the object itself. The only persistent structures required by PMS are one *Incoming Reference List* (IRL) per partition which is maintained as a persistent B^+-tree.

Our performance study in this section is based on the standard OO7 benchmark [CDN93]. In particular, we worked on the standard *small-9* dataset in OO7 which was also used in [YNY94] for their simulation study. The OO7 parameters and their values for this dataset are given in Table 1 and are explained below.

The OO7 dataset is composed of a number of *modules*, specified by NUMMODULES. Each module consists of a tree of objects called assemblies. The tree is a complete tree with a fanout of NUMASSMPERASSM and has NUMASSMLEVELS levels. The last level of the tree is called a base assembly while the upper levels are called complex assemblies. In addition, each module consists of NUMCOMPPERMODULE composite objects. The base assemblies point to NUMCOMPPERASSM of these composite objects. Many base assemblies may share a composite object.

Each composite object points to: (a) a private set of NUMATOMICPERCOMP *atomic objects*, (b) a distinguished atomic object (called the *composite root*), and (c) a *document* object. An atomic object has a fixed number of connections (specified by NUMCONNPERATOMIC) out of it, to other atomic objects in the same set. A connection is itself modeled as an object (called a *connection object*) pointed to by the source of the connection and in turn points to the destination of the connection. The connections connect the atomic objects into a cycle with chords. We will call a composite object along with its private set of atomic objects, connection objects and the document object together as an *object composite*. All object references in the benchmark have inverses and we always insert or delete references in pairs (the reference and its inverse).

The dataset consisted of 104280 objects occupying 4.7 megabytes of space. Each object composite consisted of 202 objects and had a size of 9160 bytes. During the course of experiments, the size was maintained constant by adding and deleting the same amount of data. The object manager used a buffer pool consisting

Parameter	Value
NUMMODULES	1
NUMCOMPPERMODULE	500
NUMCONNPERATOMIC	9
NUMATOMICPERCOMP	20
NUMCOMPPERASSM	3
NUMASSMPERASSM	3
NUMASSMLEVELS	7

Table 1: Parameters for the OO7 benchmark

of 500 4KB pages. The I/O cost is measured in terms of the number of 4KB pages read from or written to the disk. All the complex and base assemblies forming the tree structure were clustered together. We also clustered together all the objects created for a composite.

For PMS, the data was divided into 4 partitions; each partition fits in memory. The inter-partition references were kept very small. All the complex and base assemblies forming the tree structure were put in the same partition. Approximately one out of every 50 composites spanned partitions.

We conducted two sets of experiments, the first was based on structure modifications suggested in the OO7 benchmark while the second modifies complex assemblies. We discuss each in turn.

6.1 Structure Modifications

The workload in this experiment consisted of repeatedly inserting five object composites and attaching each composite to a distinct base assembly object, and then pruning the newly created references to the same five object composites – we call this whole set of inserts and deletes an update pass. This corresponds to the structure modification operations of the OO7 benchmark. This workload represents the case when an application creates a number of temporary objects during execution and disposes them at the end of the execution. The results presented are over 90 update passes interspersed with garbage collection; garbage collection is invoked when the database size crosses 5MB (recall the steady state database size is 4.7MB).

We first present the cumulative overheads (cost during during normal processing as well as the overhead due to the garbage collection thread) for this workload.

Metric	TCRC	PMS
Logs (MB)	143.97	110.52
I/O:Read+Write	355+53701	31033+44833

Although the amount of logs generated by the TCRC algorithm is more than that of the PMS algorithm, the overall I/O performance (including the I/O's for logs) of TCRC is about 50% better than PMS for this workload. Three factors contribute to the overall performance: the frequency of invocation of the garbage collector, the overhead during a garbage collection pass, and the overhead due to normal processing. We study these three factors in detail now.

6.1.1 Invocation Frequency

We checked the database size at the end of every update pass and invoked the garbage collector if the database size exceeded 5 MB. TCRC collects all garbage and therefore the amount of garbage, which is generated at the rate of 45800 bytes per update pass, exceeded 0.3 MB (and thus the total database size exceeded 5 MB) after seven update passes. Thus, garbage collection in case of TCRC is consistently invoked after every seven update passes.

The pattern is more interesting in the case of PMS. Approximately one out of fifty composites spanned partitions; such a composite (which is cyclic) is never collected. This caused the database size to increase with time. Since the threshold remained fixed at 5 MB, this caused the garbage collection to be invoked more frequently as time progressed. During the course of the 90 update passes, TCRC garbage collector was invoked 12 times, while PMS was invoked 14 times. Initially, the PMS collector was invoked every seven update passes, then every six update passes and by the end of the 90 update passes every five update passes. By the end of the 90 update passes, there were 73280 bytes of uncollected garbage for PMS.

6.1.2 Overhead of a Garbage Collection Pass

The table below gives the average I/O overhead and the amount of logs generated by TCRC and PMS for an invocation of the collector. To get the total cost the figures have to be multiplied by the number of invocations (which is 14 for PMS and 12 for TCRC).

Metric	TCRC	PMS
Logs (MB)	1.40	1.07
I/O:Read+Write	0+514	2007+568

Since garbage collection was invoked right after the insertions, TCRC found all the objects that it had to traverse in the cache and incurred no reads. PMS needed to make a reachability scan from the root and therefore had to visit all of the 104280 objects in the dataset. This accounts for the excessive reads incurred by PMS. The logs generated by TCRC is however bigger than PMS since (i) the size of an object is bigger (due to the presence of reference counts) and therefore the logs corresponding to the deletion of garbage objects are larger and (ii) the garbage objects are deleted from WRT and these deletions have to be logged (recall that all newly created objects will be in WRT since all new pointers are weak).

6.1.3 Normal Processing Overheads

The following table shows the amount of I/O performed and the amount of logs generated during normal processing (when the collector is not running) over the course of the 90 update passes.

Metric	TCRC	PMS
Logs (MB)	127.17	97.33
I/O:Read+Write	355+47533	2941+37274

The algorithms have to maintain the persistent data structures consistent with the data during normal processing. In the case of PMS, the only persistent data structure is the IRL which is updated quite rarely. On the other hand, in the case of TCRC, the reference counts as well as the WRT may be updated. The amounts of log generated show the additional logging that has to be performed by TCRC for maintaining these persistent structures. The additional logs account for about 8000 extra writes for TCRC. The rest of the extra writes performed by TCRC (about 2000) are due to writing parts of WRT back as a result of normal cache replacement. The amount of reads performed by TCRC is significantly smaller that PMS because the cache is not disturbed much by the garbage collection thread in the case of TCRC. In the case of PMS, at the end of the collection pass the cache could contain many objects from the assembly tree which are not required during normal processing.

6.2 Updating Complex Assemblies

In this set of experiments, we updated the assembly hierarchy tree by replacing a subtree rooted at a complex assembly by a different one. The lowest level base assemblies in the new hierarchy tree pointed to the same composite objects. In this experiment, we modified the OO7 benchmark by removing the back pointers to the base assembly objects from the composite objects. This provides acyclic data which enables us to test our schema graph optimization. It also limits the traversal of TCRC.

We varied the level of the root of the the subtree that we were replacing. The level was varied from two to six (level n corresponds to the level which is the n^{th} level upwards from the base assemblies). Notice that the subtree that was replaced is garbage after this update. After such a update we invoked the garbage collector. The higher the level of the root of the subtree being replaced, the more the number of object composites reachable, and therefore the more the number of objects TCRC had to traverse. In this experiment, we report only on the overheads of the garbage collection pass. The normal processing overheads are very similar to the previous experiment since we are creating some number of objects and pruning references to others like the previous experiment. The cost of the garbage collection phase for TCRC is tabulated below:

Metric	Level of Root of Subtree				
	2	3	4	5	6
Logs (MB)	0.00	0.01	0.05	0.16	0.49
I/O:Read	77	356	10291	21209	32388
I/O:Write	8	35	177	376	1309

The cost of the garbage collection phase for PMS is tabulated below:

Metric	Level of Root of Subtree				
	2	3	4	5	6
Logs(MB)	0.00	0.00	0.00	0.02	0.05
I/O:Read	1736	1736	1736	1737	1742
I/O:Write	10	13	18	27	31

The results show that number of reads by TCRC is smaller than the number of reads by PMS for modifications at the lower levels but degrades for modifications higher up the hierarchy. This is expected since TCRC performs a local traversal. The number of reads for PMS is the same for modifications at all levels. Notice however that even though PMS traverses the entire graph, the cost of TCRC is significantly higher than PMS for modifications higher up the hierarchy. There are two reasons for this. The first is that TCRC reads all objects as it encounters their references during the traversals unlike PMS which follows only intra-partition references. This results in excessive read overhead since there is a lot of cache conflicts for objects on different pages. Secondly, the RRT is disk resident and as its size grows, there is extra I/O overhead for accessing RRT. In contrast, our implementation of PMS assumes information about which objects in a partition have been marked during the mark phase can be maintained in memory itself.

The amount of logs generated by TCRC (a 0.00 for the amount of logs generated indicates that the amount of logs generated is less than 10KB) grows in comparison to the logs generated by PMS as the level number grows since GreenTraverse updates pointer strengths of objects, which are also logged. The more the objects traversed, the more the number of pointers whose strengths get changed. In fact, most of the information in the logs generated by the TCRC is very small (either a pointer strength update, an update to WRT or an update to the reference count). However, each of these logs has a significant log header overhead in the *Brahmā* system. In a system which can club all these logs under a single log header along with the log for the actual pointer update, the overheads will come down drastically. We are currently modifying the log subsystem in *Brahmā* to do this.

The TCRC algorithm can be optimized by using semantics available from the schema graph. Notice that the template for the pointer from a complex assembly to a base assembly is acyclic and therefore need not be traversed by the RedTraverse algorithm thus preventing TCRC from unnecessarily traversing the object composites. The cost of the TCRC garbage collection pass when the experiment was repeated with this schema-based optimization are tabulated below. It can be seen that TCRC with the optimization outperforms the basic TCRC as well as the PMS algorithm.

Metric	Level of Root of Subtree				
	2	3	4	5	6
Logs(MB)	0.00	0.01	0.02	0.06	0.17
I/O:Read	0	0	0	0	2
I/O:Write	8	9	12	27	67

7 Conclusions and Future Work

We have presented a garbage collection algorithm, called TCRC, based on cyclic reference counting and proved it correct in the face of concurrent updates and system failures. We have implemented and tested the algorithm.

Our performance results indicate that TCRC can be much cheaper, at least in certain cases, than partitioned mark-and-sweep since it can concentrate on local cycles of garbage. We believe our algorithm will lay the foundation for cyclic reference counting in database systems.

We plan to explore several optimizations of the TCRC algorithm in the future. For instance, we have observed that just after creation of the datasets, garbage collection has to perform extra work to convert weak pointers into strong pointers. However, once the conversion has been performed, a good set of strong pointers is established, and the further cost of garbage collection is quite low. It would be interesting to develop bulk-loading techniques for reducing the cost of setting up pointer strengths.

We plan to optimize RedTraverse by only following a strong pointer into an object if all other strong pointers into that object have been already encountered. This will greatly reduce the number of objects traversed and may lead to significant performance benefits. Finally, another interesting extension of the TCRC algorithm would be to develop a partitioned TCRC algorithm in which during a local mark and sweep only intra-partition edges are traversed.

Acknowledgments

We thank Jeff Naughton and Jie-bing Yu for giving us a version of their garbage collection code which provided us insight into garbage collection implementation. We also thank Sandhya Jain for bringing the work by Brownbridge to our notice.

References

[AFG95] L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Object Database Systems. In *Procs. of the International Conf. on Very Large Databases*, September 1995.

[ARS+97] S. Ashwin, Prasan Roy, S. Seshadri, Avi Silberschatz, and S. Sudarshan. Garbage Collection in Object Oriented Databases Using Transactional Cyclic Reference Counting. Technical report, Indian Institute of Technology, Mumbai, India, June 1997.

[Bro84] D.R. Brownbridge. *Recursive Structures in Computer Systems.* PhD thesis, University of Newcastle upon Tyne, United Kingdom, September 1984.

[Bro85] D.R. Brownbridge. Cyclic Reference Counting for Combinator Machines. In Jean-Pierre Jouannaud, editor, *ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 273–288. Springer-Verlag, 1985.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proc. of the ACM SIGMOD Int. Conf., Washington D.C.*, May 1993.

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

[JL91] Richard E. Jones and Rafael D. Lins. Cyclic weighted reference counting. Technical report 95, University of Kent, Canterbury, United Kingdom, December 1991.

[Lin90] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. Technical report 75, University of Kent, Canterbury, United Kingdom, June 1990.

[MWL90] A.D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[PvEP88] E.J.H. Pepels, M.C.J.D. van Eekelen, and M.J. Plasmeijer. A cyclic reference counting algorithm and its proof. Internal Report 88-10, University of Nijmegen, Nijmegen, 1988.

[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, February 1994.