Garbage Collection in Object Oriented Databases Using Transactional Cyclic Reference Counting

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

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

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

[*] Currently at University of Wisconsin Madison

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.

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. We also present a proof of correctness of the TCRC algorithm.

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.

The algorithm presented in this paper improves on that presented in an earlier extended abstract of this paper [ARS+97], in the following ways. There is no longer an assumption that transactions follow *strict 2PL*; in fact the current algorithm makes no assumptions about the locking policies used by the transaction. There is no longer an assumption that transactions follow *strict WAL* (that is, both the undo and redo values must be logged before actually performing the update); only the normal (non-strict) WAL is assumed to be followed. That is, the current algorithm requires undo values to be logged before the update, while the redo values may be logged *anytime* before the end of the transaction (before or after the update). Finally, the current algorithm performs a more restricted local traversal than the earlier algorithm, and is therefore potentially more efficient.

For the client-server setting, the current algorithm also relaxes the *force* requirement; that is, updates made by a transaction running at the client can be reflected at the server after the transaction ends, and are not required to be forced to the server before the end of the transaction.

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.

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.

As with any other garbage collection scheme, we assume that an object identifier (*oid*) is valid only if it either refers to 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 oid from an object in the database or created the object it refers to. Note that this precludes transactions from passing oids to other transactions, and from storing oids in external persistent storage.

Assumption 3.1 *The transactions follow* **WAL**, *that is, they log the undo value before actually performing the update, but the redo value may be logged anytime (before or after the update).* □

Assumption 3.2 *All logs for a transaction are forced to disk before commit or abort (force-logs-at-abort in addition to force-logs-at-commit).* □

The assumptions above are satisfied by typical storage managers for object-oriented databases.

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 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. 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 [Bro84, Bro85]. 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 another 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 after a system crash by scanning the entire database. Therefore, it is not necessary to make it persistent and incur the overhead of logging updates to these structures. Reconstructing this information at crash will however affect the availability of the database. If fast recovery is required then we could make these structures persistent at the cost of extra logging. The choice of whether or not to make this information persistent can be left to each installation.

If the above structures are made persistent, then 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. Thus their updates will be undone if the transaction does not complete, and will be redone (while repeating history) if the system crashes.

Apart from the above structures, we have the following non-persistent structures – this means that irrespective of whether the above structures are made persistent these need not be persistent.

There is 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.

Similar to [AFG95] TCRC also 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. An oid in TRT is tagged with the tid of the transaction that is reponsible for its insertion into TRT. There may be multiple entries in TRT for the same oid. 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.

4.2 The Algorithm

TCRC consists of two distinct algorithms, run by different processes. The first is the *log-analyzer* algorithm (LogAnalyzer). The second algorithm is the actual *garbage collection* algorithm (CollectGarbage). The execution of these is synchronized by two latches: a log_analyzer_latch that is taken for the duration of LogAnalyzer, and a gc_latch that is taken for the duration of CollectGarbage. We describe the two algorithms 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. We shall assume it is run as part of the transaction itself, and 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.

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 6 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. The oid of the affected object is inserted into the TRT tagged with the tid of the transaction that made the update.

- For log records corresponding to the creation of objects, the reference counts for the new object are initialized to zero, and the oid of the created object is inserted into the WRT. The oid of the created object is inserted into the TRT tagged with the tid of the transaction that created the object.

- For end-of-transaction (commit or abort) log records, the algorithm first tries to get the gc_latch. If the latch is obtained immediately, then garbage collection is not in progress and remove all the oid entries for the terminating transaction from the TRT and the gc_latch released thereafter (recall that each oid entry in TRT is tagged with the tid of the transaction that is responsible for its presence in TRT). However, if the gc_latch 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.

The following properties follow from the above discussion.

Property 4.1 The persistent root is never placed in the WRT. It has no references to it. Therefore it never occurs in the TRT or RRT. □

Property 4.2 The objects in TRT corresponding to a transaction are removed only when (a) the transaction has ended *and* (b) garbage collection is not in progress. □

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 RedTraverse populates the RRT with objects it identifies as potential garbage. The rest of the garbage collection algorithm is restricted to only the objects in RRT. The pseudocode for RedTraverse appears in Figure 1. RedTraverse performs a fuzzy localized traversal of the object graph. It is invoked on all objects in WRT that are not in TRT. Thereafter, an object is visited by any of the invocations if and only if the object is not in TRT and all the objects that have a strong pointer to this object have been visited earlier. No locks are obtained on the objects being traversed. Short term latches may be obtained on objects or pages to ensure physical consistency. RedTraverse marks all visited objects red and puts them in RRT.

Additionally, RedTraverse caches the reference counts ($SRefC$ and $WRefC$) of each object in RRT at the instant it visits the object. RedTraverse also maintains for each object in RRT, two counts: $SRedRefC$ and $WRedRefC$, giving respectively the number of strong and weak pointers to the object from all other objects visited. 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 traversal. In the pseudocode of Figure 1, these objects are coloured blue. For the rest of the paper, we ignore the presence of blue objects in RRT, and assume that they are explicitly removed from RRT after the last invocation of RedTraverse. The invocations of RedTraverse are collectively termed *red traversal*.

Procedure GreenTraverse performs a fuzzy traversal with the purpose of marking live objects in RRT green and updating some pointer strengths to ensure that every object it visits has at least one strong pointer referring to the object (this ensures that in the absence of update transactions during the garbage collection phase, no object will be in WRT thus ensuring that there is no work to be done during the next garbage collections phase. See Theorem 5.11 for a formal proof of this statement). In addition, the pointer

```
Procedure RedTraverse(oid)
Input: oid: (oid of) the object to be traversed
{
    if oid is not in RRT
        InsertRRT(oid)
    Traverse(oid)
}

Procedure Traverse(oid)
Input: oid: (oid of) the object to be traversed
{
    if oid is not in TRT {
        colour oid red
        for each pointer oid → poid {
            if poid is not in RRT
                InsertRRT(poid)
            update SRedRefC_poid and WRedRefC_poid
                depending on the strength of oid → poid
            if SRedRefC_poid == SRefC_poid
                /* this is the last strong pointer to poid */
                Traverse(poid)
        }
    }
}

Procedure InsertRRT(oid)
Input: oid: (oid of) the object to be inserted into RRT
{
    colour oid blue
    insert oid into RRT
        with SRedRefC_oid = WRedRefC_oid = 0
    cache current values of SRefC and WRefC for oid
        in SRefC_oid and WRefC_oid respectively
}
```

Figure 1: Pseudo Code for RedTraverse

```
Procedure GreenTraverse(oid)
Input: oid: (oid of) the object to be traversed
{
    colour oid yellow
    for all pointers oid → poid {
        if poid is in RRT and is not yellow
            GreenTraverse(poid)
    }
    for all pointers oid → poid {
        if poid is green {
            if SRefC_poid == 0
                remove poid from WRT
            make oid → poid strong (if weak)
            get log_analyzer_latch
            update reference counts of poid
            release log_analyzer_latch
        } else {
            make oid → poid weak (if strong)
            get log_analyzer_latch
            update reference counts of poid
            release log_analyzer_latch
            if SRefC_poid == 0
                insert poid into WRT
        }
    }
    colour oid green
}
```

Figure 2: Pseudocode for GreenTraverse

```
Procedure CollectGarbage
{
    acquire gcLatch
    /* P_0 */
    RRT = {}
    for each oid in WRT but not in TRT
        RedTraverse(oid)
        /* also caches the reference counts
            of visited objects in SRefC_oid and WRefC_oid*/
    /* P_1 */
    TLIST = list of all transactions active at P_1
    Wait for all transactions in TLIST to end
    /* P_2: instant when all transactions in TLIST end */
    for each oid in RRT but not in TRT {
        /* SRefC_oid and WRefC_oid refer to
            cached reference counts */
L_0 :   if SRefC_oid + WRefC_oid >
                SRedRefC_oid + WRedRefC_oid {
            if SRefC_oid == 0 /* oid is in WRT */
                remove oid from WRT
            get log_analyzer latch
            invert the strength of all references to oid
            update reference counts of oid
            release log_analyzer latch
            GreenTraverse(oid)
        }
    }
    /* P_3 */
L_1 : for each oid in RRT that is red and is in TRT
        GreenTraverse(oid)
    /* P_4 */
    for each oid in RRT that is red
        Collect(oid)
    /* P_5 */
    release gcLatch
    remove all flagged entries from TRT
}
```

Figure 3: Pseudo Code for CollectGarbage

strength updates have to be done in such a fashion that strong cycles do not remain at the end of the garbage collection phase. The pseudocode for GreenTraverse appears in Figure 2. Starting from the object Procedure GreenTraverse is invoked on, it visits all the objects in RRT that are reachable from this object in a depth first manner: it backtracks from an object after it has visited all objects in RRT reachable from that object. Just before backtracking from an object, it colours the object green and updates (if needed) the strengths of the references out of the object. If the reference is to a green object, it is made strong if it is weak; otherwise if the reference is to any other object (possibly to an object not in RRT) then it is made weak if it is strong. The invocations of GreenTraverse are collectively termed *green traversal*.

Procedure Collect actually deletes an object; before doing so, it deletes all pointers out of the object, updating the stored reference counts ($SrefC$ and $WRefC$) of the objects pointed to. It also removes the object from WRT.

The garbage collection algorithm is implemented by Procedure CollectGarbage, shown in Figure 3. Below, we present a discussion of the steps involved. Additionally, we point out instances during the execution (shown in the figure) that will be referred in the proof.

The first step is to acquire gc_latch. At some point P_0 after this, RedTraverse is invoked on all objects that are in WRT but not in TRT. After the red traversal, we wait for all transactions that were active at some point P_1 after the end of the red traversal to terminate.

This wait is necessary for correctness of the algorithm in face of transactions following (non-strict) WAL and arbitrary locking protocols. We present the intuition below; the formal proof appears in Section 5.

In case a transaction T does not follow strict-2PL locking, the following scenario can occur. T takes a lock on some object x, reads the reference to y from some pointer field in x, and then releases the lock on x. Next, all the references to y get deleted and all the transactions that made the deletions commit (this can happen because T has released the lock on x). This makes y unreachable from any live object; but y is not garbage because oid of y is cached by T which can insert a pointer to y from some other object later. The wait ensures that all transactions such as T end before analysis proceeds.

The wait further ensures, in face of (non-strict) WAL, that all the redo logs for pointer inserts that occured during the red traversal are forced to disk in addition to the undo logs for pointer deletions before analysis proceeds; this is because *all* the logs for a transaction are (by assumption) necessarily forced before the end of the transaction. This makes the TRT consistent with respect to any insertion or deletion of pointers that might have occured during the red traversal.

The list of transactions TLIST can be determined fuzzily — that is, we need not take a latch on the transaction table while scanning it. This is safe because only the following may happen while the scan is in progress: (a) some transaction that was active at P_1 ends and does not appear in TLIST — this is acceptable because we were just going to

wait for it to end anyway; or (b) some some transaction starts after P_1 and its entry appears in TLIST — this is acceptable because this can only extend the wait. P_2 is the instant when all the transactions in TLIST terminate.

Next, we do green traversal to mark green all the live nodes in RRT. After P_2, GreenTraverse is invoked on an object that is in RRT but not in TRT if the total red reference counts ($SRedRefC + WRedRefC$) for the object computed during the red traversal is strictly less than its total reference counts ($SRefC + WRefC$) that are cached during the red traversal (as checked in statement L_0). As will be shown in the proof, these objects are live: they are referred from some object not in RRT. But before the invocation, the strengths of all references to this object are inverted and the reference counts updated atomically with respect to the log-analyzer (log_analyzer_latch is used for this purpose). The inversion of strengths is necessary to ensure that after this garbage collection phase is over, there will be at least one strong pointer to the object (notice that all references to the object from objects not in RRT must be weak). This is necessary to ensure that in absence of update transactions no work will be done by the next garbage collections phase; this is formally proved in Theorem 5.11. This inversion of strengths might cause strong cycles to be formed. But, as proved in Lemma 5.8, these will not exist after this garbage collection phase is over.

After the above invocations complete at P_3, any objects in RRT that are in TRT are also marked green since their references may still be stored in an ongoing transaction and can potentially be stored back in the database. Objects that are reachable from the above objects are also marked green, by invoking GreenTraverse. These invocations get over at P_4.

In the pseudocode, we have left unspecified how (at step L_1) the consistent point is obtained such that at this point, no red object in RRT is in TRT. It turns out that the following simple procedure is enough. We make repeated scans of the RRT, invoking GreenTraverse on red objects that have been inserted into TRT since they were checked in the previous scan; and terminating when we come across no such object in the latest scan. The consistent point corresponds to the instant the last scan starts. This is because if some object in RRT is red and is in TRT at the start of the last scan, then during the scan it stays red because of the fact that GreenTraverse is not invoked in the interim; and stays in TRT because of Property 4.2(b). But then, it must be detected to be red and in TRT during the last scan — a contradiction.

All objects in RRT that are red at P_4 are collected next. The collection gets over at P_5. Finally, the gc_latch is released and all the entries in TRT that were tagged as removable by transactions that completed since the garbage collection phase began (that is, since gc_latch was acquired) are removed.

4.2.3 Support for Logical Undo by the Recovery Manager

The TCRC algorithm needs some support from the recovery manager in the form of supporting logical undos to ensure correctness. This support is required only if we choose to maintain the reference counts, pointer strengths and WRT persistent. 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.

5 Proof of Correctness

We formally state the definition of a garbage object.

Definition 5.1 (Garbage Object) An object is defined to be garbage if it is not reachable from the persistent root or from any object in the TRT or from any object whose reference (oid) has been read by any active transaction. □

The above definition considers an object as live if it is reachable from TRT, even if it does not satisfy the other conditions and therefore is garbage in the conventional sense. Note that eventually an object that is garbage in the conventional sense will leave TRT and any active transaction that read a reference to it will terminate and thus will become garbage in the sense of Definition 5.1. Our lemmas and proofs are simplified by using the above definition.

5.1 Object States

At any instant between P_1 and P_4, an object x in RRT is in one of the following three states:

S_1: x is reachable from an object outside RRT but is not reachable, through a path consisting only of objects in RRT, from any object that is in RRT and also in TRT.

S_2: x is reachable, through a path consisting entirely of objects in RRT, from an object in RRT that is also in TRT.

S_3: x is neither in state S_1 nor in state S_2, that is x is neither reachable from any object outside RRT nor from any object in RRT that is also in TRT.

In the above, an object is assumed to be reachable from itself through a null path.

We need to prove that TCRC is *safe*: it does not collect any live objects; and *complete*: it eventually collects all garbage objects.

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 guarantee that this does not happen in TCRC; that is, in the absence of update transactions, the system eventually reaches a state where garbage collection thread does no more work.

We will make use of the following properties of the algorithm in the proof of the results that follow.

Lemma 5.1 *If a transaction not in TLIST active at an instant P strictly between P_1 and P_2 has read a reference to an object that is in state S_3 at P, then the object is in state S_2 at P_2.*

Proof: Suppose some transaction T not in TLIST is active at P and has read a reference to some object that is in state S_3 at P but not in state S_2 at P_2. Let x be the first such object to which a reference is read by T. Also, let P' be the instant when T reads the first object y that has a reference to x.

The reference from y to x was present at P'. Either of the following two cases are possible:

Case 1: The reference from y to x is present at P.

Since x is in state S_3 at P, y must also be in state S_3 at P. Definitely, a reference to y was read by T before the reference to x. Recall that x was the first object that is in state S_3 at P but not in state S_2 at P_2 a reference to which was read by T. Therefore, y is in state S_2 at P_2. If the reference from y to x is present at P_2, then x is in state S_2 at P_2. Otherwise, if the reference from y to x has been deleted between P and P_2, then by Assumption 3.1 and Property 4.2(b) x is in TRT, and hence in state S_2, at P_2.

Case 2: The reference from y to x is not present at P.

The reference from y to x has been deleted between P' and P. Because of Assumption 3.1, the log for the above deletion must have been analyzed between P' and

P. P' occured after P_1 because T could not have been active at P_1. Therefore, by Property 4.2(b), x must be in TRT, and hence in state S_2, at P.

The above implies that x must be in state S_2 at P_2. Proved by contradiction. □

Lemma 5.2 *If an object is in state S_3 at an instant P between P_2 and P_4, then no transaction active at P could have read a reference to it.*

Proof: Suppose some transaction T is active at P and has read a reference to some object that is in state S_3 at P. Let x be the first such object to which a reference is read by T. Also, let P' be the instant when T reads the first object y that has a reference to x.

Definitely, a reference to y was read by T before the reference to x. Recall that x was the first object in state S_3 a reference to which was read by T. Therefore, y is not in state S_3 at P. Since x is in state S_3 at P, the reference from y to x is not present at P. But it was present at P'.

The reference from y to x has been deleted between P' and P. Because of Assumption 3.1, the log for the above deletion must have been analyzed between P' and P. P' occured after P_1 because T could not have been active at P_1. Therefore, by Property 4.2(b), x must be in TRT at P, and therefore cannot be in state S_3 at P. Proved by contradiction. □

Lemma 5.3 *If an object v is in RRT but is not in TRT at P_2, then (a) no reference to v is updated (inserted or deleted) between P_0 and P_1; and (b) all references to v that exist between P_0 and P_1 are accounted in the total reference counts of v cached during red traversal.*

Proof: No log corresponding to an update of a reference to v that occurs before P_1 is analyzed after P_0. This is because the transaction that is responsible for the update must end before P_2. Therefore, by Assumption 3.2, the log must be analyzed before P_2. But then, by Property 4.2(b), v would be in TRT at P_2 — a contradiction.

Suppose an update of some reference to v takes place at some instant P between P_0 and P_1, consider the instant when the log for the reference update is analyzed. As shown above, this must occur before P_0. But then, because the transaction is active at P by Property 4.2(a) v is in TRT at P, and therefore is in TRT at P_2 by Property 4.2(b), a contradiction. This proves part (a) of the lemma.

All references to v that exist between P_0 and P_1 are accounted in the total reference counts of v at P_0. This is because otherwise it must be that the log for the insertion of the unaccounted reference is analyzed at some instant after P_0, a contradiction. The total reference counts for v must have remained unchanged between P_0 and the instant when they are cached during the red traversal, again because of the fact that no log is analyzed in the interim that can cause the change. This implies part (b) of the lemma. □

Lemma 5.4 *If an object in RRT is in state S_1 at P_2, then it is reachable from some object not in RRT at P_0.*

Proof: Suppose that an object x in state S_1 at P_2 is not reachable from any object not in RRT at P_0. Now, consider the instant P when the first path to x from some object not in RRT was created. Let the insertion of the reference from object y to object z that is in RRT be responsible for the same. Then, at P, there exists a path from z to x consisting entirely of objects in RRT. This path remains intact till P_2 because otherwise x would be in state S_2 at P_2 by Assumption 3.1 and Property 4.2(b), which is a contradiction.

If the transaction T that made the above insertion existed before P_1, then it would end before P_2 and therefore because of Assumption 3.2 and Property 4.2, z would be in TRT at P_2. But then x would be in state S_2 at P_2 leading to a contradiction. Thus, T started after P_1.

This implies that P occured after P_1. At some instant P' between P_1 and P, T must have read a reference to z. But since z is unreachable from any object not in RRT or from any object in TRT at P', it must be in state S_3 at P'. But then, by Lemma 5.1, z (and hence x) must be in state S_2 at P_2, leading to a contradiction. □

The following result states the restrictions on the state transition of objects during garbage collection.

Lemma 5.5 *If an object in RRT is in state S_3 at some instant P between P_2 and P_4, then it remains in state S_3 between P and P_4.*

Proof: Suppose that an object x in state S_3 at some instant P between P_2 and P_4 makes a transition to some other state immediately after P.

Let the set A contain all the objects from whom x is reachable at P. By definition of state S_3, no object in A lies outside RRT. Moreover, at P no object in A is reachable from an object outside RRT or from an object in RRT that is also in TRT — otherwise x would also be reachable from this object at P and therefore not be in state S_3. This implies that all objects in A are in state S_3 at P.

The transition of state must be due to an update of a reference to some object y in A; the instant P corresponds to the occurence of the update itself, or the analysis of the log for the update — whichever is earlier. The transaction T that is responsible for the update must have obtained a reference to y before it is able to make the update or generate a log for the same. In other words, T must have obtained a reference to y before P. Because of Assumption 3.2, T is active at P. But this contradicts Lemma 5.2 because y is in state S_3 at P as shown above. Proved by contradiction. □

Next, we prove an invariant of the algorithm.

Lemma 5.6 *At P_4, an object in RRT is red iff it is in state S_3.*

Proof: **(Only if)** Suppose that at P_4, an object x in RRT is red and is not in state S_3. By Lemma 5.5, x is not in state S_3 at P_2.

Consider the instant P when the condition in the statement L_1 evaluates to false leading to termination of the for-loop. At P, therefore, all objects in RRT that are also in TRT are green. Two cases are possible.

Case 1: x is in state S_1 at P_2.

Then, by Lemma 5.4, it is reachable from some object not in RRT at P_0. Let the reference from object u to object v be along the path such that u is not in RRT but v is in RRT. Now, two subcases are possible.

Case 1.1: The path from u to x is intact between P_0 and P.

By Lemma 5.3(b), the reference from u to v as well as the references that are accounted in the red reference counts of v are accounted in the cached total reference counts of v. Since u is not in RRT, the reference from u to v is not accounted in the red reference counts of v. Thus, the cached total reference counts of v are strictly greater than the red reference counts of v.

Since v is in RRT but not in TRT, statement L_0 will be executed for v. For the reasons stated above, the condition will be satisfied and GreenTraverse will be invoked on v. Since the path from v to x consists of only objects in RRT and is intact at this point, x will be coloured green (if not already so).

Case 1.2: The path from u to x is broken between P_0 and P.

Consider the object y along the path at P_0 such that the reference to y along the path is deleted between P_0 and P, but the path from y to x is intact at P (y might be the same as x). Being in TRT, y must be green at P. But since the path from y to x is intact between P_0 and P, the invocation of GreenTraverse that coloured y green must also have coloured x green (if not already so).

Case 2: x is in state S_2 at P_2.

That is, at P_2 there existed a path, consisting only of objects in RRT, to x from some object in RRT that is also in TRT. The path could have broken between P_2 and P. The rest of the proof is similar to Case 1.2 above. Consider the object y along the path at P_2 such that the reference to y along the path is deleted between P_2 and P, but the path from y to x is intact at P (y might be the same as x). Being in TRT, y must be green at P. But since the path from y to x is intact between P_2 and P, the invocation of GreenTraverse that coloured y green must also have coloured x green (if not already so).

The above implies that x is green at P, and therefore at P_4 — a contradiction. This proves that x must be in state S_3 at P_4. □

Proof: **(If)** Suppose that a green traversal invoked at some object x was responsible for colouring green some object y that is in state S_3 at P_4.

The path from x to y that existed at the time of the green traversal must be intact at P_4 — otherwise y would be in state S_2 at P_4 by Assumption 3.1 and Property 4.2(b). Therefore, x must also be in state S_3 at P_4. In particular, x could not have been in TRT at the time of the invocation.

But then, the only way green traversal could have been invoked on x is that the condition in statement L_0 must have evaluated to true when it was executed for x. That is, the red reference counts for x are strictly less than the cached total reference counts for x. This can only occur if a reference from some object z to x that existed (as per the cached total reference counts of x, by Lemma 5.3(b)) between P_0 and P_1 was not traversed in the red traversal and therefore is not accounted in the red reference counts of x.

By Lemma 5.3(a), no update of any reference to x takes place between P_0 and P_1. This implies that the reference from z to x existed at P_0. Moreover, this reference could not have been deleted between P_0 and P_4, otherwise by Assumption 3.1 and Property 4.2 x would be in TRT at P_4, a contradiction. This further implies that z must be in RRT, otherwise x would not be in S_3 at P_4.

Now, we know that z was visited by the red traversal between P_0 and P_1. Also, the reference from z to x existed at P_0 and it did not get deleted between P_0 and P_1. But then, the reference would have been accounted in the red reference counts of v — a contradiction. □

The results stated above are put together in the form of the following theorem.

Theorem 5.7 (Safety) Only garbage objects are collected by CollectGarbage.

Proof: At P_4, all red objects are in state S_3 by Lemma 5.6. By definition of the state S_3, these objects are not reachable from (a) any object not in RRT. This implies that they are not reachable from the persistent root (which is never in RRT by Property 4.1) or from any object in TRT not in RRT; (b) any object in TRT that is in RRT. Therefore, these object are neither reachable from the persistent root nor from any object in TRT.

All objects from which an object in state S_3 is reachable at P_4 are in state S_3 at P_4. By Lemma 5.2, no reference to any of these objects has been read by any transaction active at P_4.

This implies, by definition, that all objects that are red at P_4 are garbage. Since only these objects are collected by CollectGarbage, the theorem is proved. □

Next, we prove that TCRC is *complete* — that is, it collects all garbage eventually. For this, we further need the following results.

Lemma 5.8 *A cycle of strong references can exist only between P_2 and P_5.*

Proof: CollectGarbage changes the reference strengths only between P_2 and P_5. Transactions can only delete strong references — they never change pointer strengths or insert

strong pointers. We assume that there exist no cycles of strong references when the objects are loaded. Therefore, it is sufficient to prove that if there exist no cycles of strong references at P_2 then there exist no cycles of strong references at P_5.

Only the strengths of references from objects in RRT are changed between P_2 and P_5. Since there are no cycles of strong references at P_2, we cannot have a cycle of strong references that does not contain an object in RRT at P_5.

Consider two green objects x and y such that the green traversal backtracked from x before it backtracked from y. Then, all references from y to x are made strong and from x to y are made weak during the traversal. Thus, at P_5 there exist no cycles of strong references containing only green objects.

Also, the green traversal makes all references from green objects to objects not in RRT weak. This further guarantees that there exist no cycles of strong references at P_5 must have red objects.

But all red objects at P_4 are collected as garbage before P_5. This proves that there exist no cycles containing strong references at P_5. □

Lemma 5.9 *If an object is garbage P_0 then it is in state S_3 at P_4.*

Proof: Let A be the set of all garbage objects at P_0. We fix an order on the objects in A such that an object comes after all objects that have a strong reference to it at P_0. This is possible because by Lemma 5.8, cycles of strong references cannot exist at P_0.

Recall that red traversal puts in RRT all objects in WRT that are not in TRT. Thereafter, it puts an object in RRT if it is not in TRT and all objects which have strong references to this object are in RRT.

Let x be the first object in the above ordering that is not in RRT at P_2. There can be two cases possible.

Case 1: There do not exist any strong references to x at P_0. But then x is in WRT at P_0 and not in TRT because it is garbage. Being garbage, it remains that way during the course of the red traversal, and therefore RedTraverse must be have been invoked on it. Thus, it must be included in RRT.

Case 2: There exist strong references to x at P_0. Since x is garbage at P_0, all objects that have a strong reference to x must be garbage at P_0. But then, they must be in A, and must occur before x in the ordering. By the choice of x, all these objects must be in RRT at P_2. Also, x is not in TRT because it is garbage. Therefore, it must be included in RRT during the red traversal.

The above implies that x must be in RRT at P_2 — a contradiction. This proves that all objects in A are in RRT at P_2. Further, none of these objects (which are garbage at P_0) are reachable from objects not in RRT at P_2 (which are live at P_0). Moreover, none of

these objects, being garbage also at P_2, are reachable from objects that are in TRT at P_2. Putting the above together, we see that all objects in A are in state S_3 at P_2.

But then, by Lemma 5.5, all objects in A must be in state S_3 at P_4. This proves the lemma. □

Theorem 5.10 (Completeness) All garbage objects are eventually collected by CollectGarbage.

Proof: By Lemma 5.9 and Lemma 5.6, a garbage object will be coloured red at P_4 in the first CollectGarbage invoked after it became garbage and therefore will be collected. □

Theorem 5.11 (Bounded Work) If TRT is empty at P_0 then in absence of any concurrently executing update transactions, WRT will be empty at P_5.

Proof: Since there are no transaction updates executing concurrently with the garbage collection, the strength of all references remains same during the course of garbage collection.

First, we consider objects not in RRT. If an object x not in RRT that has a strong reference from some object in RRT, then x must have at least one strong reference from some object not in RRT — otherwise x would have been included in RRT during red traversal. Since this strong reference is not traversed by during the green traversal, it is never made weak by the garbage collector. Further, since only objects in RRT get deleted, the reference never gets deleted either. This implies that any object that is not in RRT is not in WRT at P_5.

We consider the objects in RRT next. The green traversal makes the references from a traversed object to as yet untraversed objects in RRT strong before traversing the latter. Thus, all green objects except the ones on which the traversal is invoked have at least one strong reference to them from other green objects at P_4.

Because no object is in TRT, green traversal is started only on objects which have references from outside RRT. By construction, all these references are weak. Therefore, when the strengths of references to an object are inverted before invocation of green traversal on it, the references to it from objects not in RRT become strong. Because these references are never traversed during the green traversal, they are strong at P_4.

Summarizing, every green object has at least one strong reference to it from another green object or from an object not in RRT at P_4. Between P_4 and P_5, red objects are collected, resulting in deletion of references from red objects to green objects. But from the results proved above, these reference deletions cannot put any green object into WRT. Thus, no green object is in WRT at P_5.

Thus, no green object or object not in RRT is in WRT at P_5. But this accounts for all the objects because there do not exist any red objects at P_5. Thus, it is proved that no object is in WRT at P_5 in absence of transaction updates. □

In particular, if no update transactions exist from the beginning of one invocation of CollectGarbage to the end of the next, then TRT will be empty at the beginning of the latter invocation; and therefore by Lemma 5.11 the WRT will be empty at the end of the second invocation.

6 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.

7 Extension to a Client Server Environment

In this section, we outline the set of assumptions required for our algorithm to work correctly in a data shipping client server environment.

Assumption 7.1 *The transactions run only at the clients. The server can determine what transactions are are possibly active at the clients at any given instant.* □

Assumption 7.2 *Cache consistency is guaranteed among the clients. That is, the transactions running at any client always see the latest state of the database.* □

This allows us to think of the transactions as running on a *single* client.

Assumption 7.3 *All undo records are received the server before the update is reflected at the server (WAL).* □

Assumption 7.4 *All logs for a transaction are received at the server before commit or abort (force-logs-at-abort in addition to force-logs-at-commit).* □

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.

To guarantee that the algorithm works correctly in the client-server setting with the above assumptions, the only change required in the algorithm is the following generalization of Property 4.2.

Property 7.1 The objects in TRT corresponding to a transaction are removed only when (a) the transaction has ended; (b) all updates by the transaction are reflected at the server; *and* (c) garbage collection is not in progress at the server. □

Note that the LogAnalyzer as well as the CollectGarbage algorithms are run at the server. As such, some extra care has to be taken during traversals of the object graph. Because the database state at the server (where the garbage collector is running) is not current, it might happen that an object may have a reference to some newly created object that is not yet present at the server. Such a dangling reference is simply ignored during the traversals.

We can prove the correctness of the algorithm in the client server setting also and refer the reader to [RSS+98] for a proof.

8 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 and has 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 an optionally persistent extendible hash table indexed on the oid while the TRT is an in-memory hash table indexed separately on the oid and the tid (to allow easy deletion of all entries of a transaction). The reference counts $SRefC$ and $WRefC$ are stored in an optionally persistent on-disk hash table. 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. Figure 4 illustrates the OO7 benchmark.

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

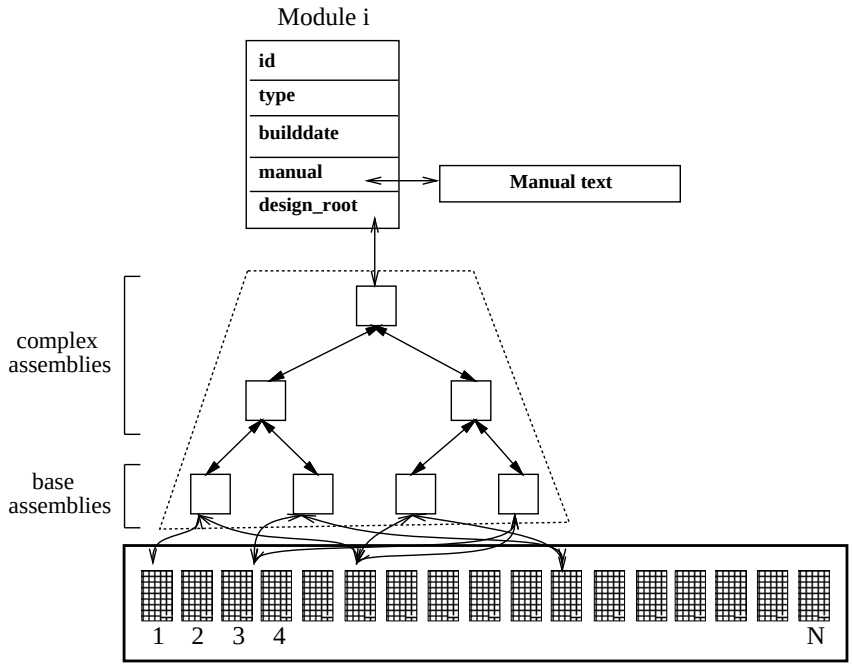

Figure 4: The OO7 Benchmark

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

Table 1: Parameters for the OO7 benchmark

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 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.

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.

As pointed out earlier, the option to have the data structures persistent (updates logged so that the information does not have to be regenerated at system start at the cost of availability) is left with the user. As such, we present the results for TCRC with logging (denoted w/logging) and without logging (denoted w/o logging) of the updates to WRT,

reference counts and pointer and object strengths. Recall that in PMS. the only data structures used are an IRL (B^+ tree) per partition which store inter-partition reference information. Since the inter-partition references are rare, there is no significant difference in the cases when these B^+ trees are persistent (updates logged) or not. Therefore, below we present only the results for PMS without logging of the updates to the IRLs.

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 of the experiments in turn.

8.1 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 w/logging	TCRC w/o logging	PMS w/o logging
Logs (MB)	150.69	113.44	113.18
I/O:Read+Write	2574+55745	2591+44111	31026+45682

Although the amount of logs generated by the TCRC algorithm with logging is more than that of the PMS algorithm, the overall I/O performance (including the I/O's for logs) of TCRC is better than PMS for this workload. However, if the logging is turned off then TCRC performs much better than PMS in terms of I/O and generates slightly more logs. The additional logs generated by TCRC include those for the extra garbage collected by TCRC.

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.

8.1.1 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.

8.1.2 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 w/logging	TCRC w/o logging	PMS w/o logging
Logs (MB)	1.69	1.07	1.07
I/O:Read+Write	33+626	12+403	1869+566

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 amount of logs generated by PMS, however, are not constant over the 14 invocations. This is again because as cyclic garbage goes on accumulating, garbage collection in PMS gets invoked more and more frequently on less and less garbage. Thus, the amount of garbage collected per collection in PMS decreases with time. This is the only update that is logged in PMS (recall that we do not log updates to IRLs). Thus, the amount of logs generated by PMS decreases with time. Thre is a corresponding decrease in the write counts too. In the table above, for comparability sake, we have presented the result for PMS as the average for the initial invocations of garbage collection which collect all

the garbage.

The log generated by TCRC for logging is however bigger than PMS since 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).

8.1.3 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 w/logging	TCRC w/o logging	PMS w/o logging
Logs (MB)	130.34	100.61	100.31
I/O:Read+Write	2404+48189	2421+39304	3017+38033

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.

In case of TCRC without logging, the amount of the logs generated is more or less the same as PMS. This is expected since exactly the same updates are logged in both cases. The amounts of log generated for TCRC with logging show the additional logging that has to be performed by TCRC for maintaining the persistent structures. The additional logs account for about 7500 extra writes for TCRC. The rest of the extra writes performed by TCRC (about 1500) are due to writing parts of WRT back as a result of normal cache replacement (these are also reflected in the results for TCRC without logging). 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 (for instance) many objects from the assembly tree that are not required during normal processing.

8.2 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.

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 with logging is tabulated below:

	Level of Root of Subtree				
Metric	2	3	4	5	6
Logs (MB)	0.00	0.01	0.02	0.09	0.28
I/O:Read	0	1	1	173	943
I/O:Write	2	7	9	67	158

The results for TCRC without logging are as follows:

	Level of Root of Subtree				
Metric	2	3	4	5	6
Logs (MB)	0.00	0.00	0.00	0.02	0.05
I/O:Read	0	1	1	173	943
I/O:Write	1	2	3	41	76

The cost of the garbage collection phase for PMS without logging is tabulated below:

	Level of Root of Subtree				
Metric	2	3	4	5	6
Logs(MB)	0.00	0.00	0.01	0.02	0.05
I/O:Read	1737	1737	1737	1738	1743
I/O:Write	10	12	19	27	31

The results show that number of reads by TCRC is much smaller than the number of reads by PMS, especially for modifications at the lower levels. This is expected since TCRC performs a local traversal while PMS necessarily traverses the whole database (except the garbage, which is small) for modifications at any level.

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 5KB) grows as the level number grows because of larger amount of garbage collected. The growth is more prominent in case of TCRC with logging in comparison to PMS since it also logs changes made to the pointer strengths during the green traversal. The more the objects traversed, the more the number of pointers whose strengths get changed, and therefore the more the logs.

The TCRC algorithm can be optimized by using semantics available from the schema graph. To illustrate the effect of this optimization, 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 limits the traversal of TCRC: the template for the pointer from a base assembly object to a composite object becomes acyclic on removal of the back pointer from the composite object to the base assembly object, and therefore need not be traversed during red traversal — thus preventing TCRC from unnecessarily traversing the object composites. The cost of TCRC with logging when the experiment was repeated with this schema-based optimization is tabulated below. It can be seen that TCRC with the optimization outperforms the basic TCRC as well as the PMS algorithm, particularly for updates at higher levels.

	Level of Root of Subtree				
Metric	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

9 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.

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. In *Procs. of the International Conf. on Very Large Databases*, pages 366–375, August 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.

[RSS+98] Prasan Roy, S. Seshadri, Avi Silberschatz, S. Sudarshan, and S. Ashwin. Garbage Collection in Object Oriented Databases Using Transactional Cyclic Reference Counting. Technical Report, Indian Institute of Technology, Mumbai, India, January 1998.

[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.