On-line Reorganization in Object Databases

Mohana K. Lakhamraju
University of California,
Berkeley CA
mohan@cs.berkeley.edu

Rajeev Rastogi, S. Seshadri
Bell Labs, Murray Hill, NJ
rastogi@research.bell-labs.com
seshadri@research.bell-labs.com

S. Sudarshan
Indian Institute of Technology,
Bombay, India
sudarsha@cse.iitb.ernet.in

Abstract

Reorganization of objects in an object databases is an important component of several operations like compaction, clustering, and schema evolution. The high availability requirements (24×7 operation) of certain application domains requires reorganization to be performed on-line with minimal intereference to concurrently executing transactions.

In this paper, we address the problem of on-line reorganization in object databases, where a set of objects have to be migrated from one location to another. Specifically, we consider the case where objects in the database may contain physical references to other objects. Relocating an object in this case involves finding the set of objects (parents) that refer to it, and modifying the references in each parent. We propose an algorithm called the Incremental Reorganization Algorithm (IRA) that achieves the above task with minimal interference to concurrently executing transactions. The IRA algorithm holds locks on at most two distinct objects at any point of time. We have implemented IRA on Brahma, a storage manager developed at IIT Bombay, and conducted an extensive performance study. Our experiments reveal that IRA makes on-line reorganization feasible, with very little impact on the response times of concurrently executing transactions and on overall system throughput. We also describe how the IRA algorithm can handle system failures.

1 Introduction

Globalization requires that corporate information systems be available twenty four hours a day, seven days a week (24×7 operations). Very high availability of database systems is also required for mission-critical applications such as telecommunications, process monitoring systems etc. For example, telecom switches typically have down time requirements of atmost three minutes in a year. A major technical challenge for architects of such highly available systems is to devise and implement *on-line* utilties for periodic and routine maintenance of the systems [ZS98, Edi96].

In this paper, we consider the problem of on-line reorganization in object databases, in which a set of objects have to be migrated from one location to another with minimal interference to concurrently executing transactions. On-line reorganization is a fundamental component of many utility operations such as:

Compaction: Continuous allocation and deallocation of space for variable length objects can result in fragmentation. Compaction gets rid of fragmentation by migrating objects to a different location and packing them closely [NOPH92].

Copying Garbage Collection: One approach to garbage collection is to copy all the live objects from a given region to a new region and then reclaim the given region [NOPH92, YNY94].

Clustering and Partitioning: The clustering of related objects within the same disk block or adjacent disk blocks greatly improves the performance of a transaction that accesses those set of objects within a small time frame. The partitioning of objects across several disks (also referred to as declustering) to enable concurrently accessed objects to be fetched in parallel can also enhance performance significantly. Based on changes in workload and updates to objects, new clustering and partitioning decisions are made, which require the system to migrate a set of objects from one location to another [WMK94, TN91].

Schema Evolution: Schema Evolution could cause an increase in object size. Such objects may have to be moved since they no longer fit in their current location. This requires reorganization of objects [BKKK87].

Performing the above operations in an online fashion requires that object references be maintained consistently while a set of objects are relocated from one place to another. Object references could be logical or physical and we consider the two cases in turn.

On-line reorganization is quite simple if the object references are logical. In this case, migrating an object O does not require the object references to O in other objects to be modified. Instead, it suffices to update the data structure that contains the mapping from the logical reference of an object to the physical location of the object. Thus, the objects can be migrated one at a time, with minimal interference to concurrent transactions, by locking out other concurrent transactions from the above mapping for an object while it is being migrated.

However, logical references typically entail one extra level of indirection for every access of the object. In disk resident databases, this could result in an additional I/O for every object access. In a memory resident database, this increases the access path length to an object by a factor of two, and may also increase main-memory requirements considerably. These overheads are unacceptable in a number of scenarios such as call setup in telecommunications, which require

response times to be in the order of tens of microseconds [JLR+94].

On the other hand, if object references are physical, they point to the actual disk or memory address of the object. Consequently, since physical object references result in more direct access to data and shorter access path lengths, main-memory database systems like DataBlitz [JLR+94, BLR+97] and TimesTen use physical instead of logical references [1]. Physical object references, however, complicate object migration since migrating an object O requires finding the set of referencers of O, $\mathcal{R}_O$, and updating the references in each object in $\mathcal{R}_O$. To find the set $\mathcal{R}_O$, we could maintain back pointers from every object in the database. However, doing so increases storage overheads greatly, and causes lock contention in back pointer lists of "popular" objects, which are pointed to from many objects. Thus, maintaining back pointers is unacceptable in many applications.

Another alternative for finding $\mathcal{R}_O$ is to traverse the object graph, and find parents of objects that need to be migrated. Performing a consistent traversal of the object graph concurrently with ongoing transactions is a non-trivial task. The naive way of doing so is to block out all transactions and perform the traversal on a quiescent database. Since this has been the only alternative available, conventional wisdom says object migration can be very disruptive to normal processing if physical references are used. This is perhaps the most important reason for the use of logical object identifiers by some vendors.

In this paper, we address the problem of on-line reorganization using object graph traversals, in an object database where references are physical[2]. Prior work [KW93] on on-line reorganization of an object-oriented database with physical references requires an action-quiescent state and some low level support from the hardware and the operating system. In addition, the proposed algorithms use forwarding addresses and require use of a complicated failure recovery technique. The problem of reorganization has been studied extensively in the context of relational databases [ZS98, Edi96, ZS96a, SD92, Omi96, AON96]. Although the references in a relational database are physical, they are stored only in the index structures. Thus, discovering the set of references to a record is a much simpler task in relational databases. Research into reorganization in relational databases has mainly concentrated on minimizing the number of locks being held and the amount of I/O necessary for reorganization. The work most closely related to ours is work on on-line garbage collection [YNY94, AFG95, ARS+97]. Though not the subject of this paper, our reorganization algorithm is also capable of performing garbage collection using an approach similar to that of a copying garbage collector [NOPH92, YNY94]. Thus, our algorithm can perform both garbage collection and reorganization and yet allow references to be physical, an ability that to the best of our knowledge, no previous algorithm in the literature possesses. We explore this and other relationships to related work in detail in Section 4.6 and Section 6.

[1] The original motivation for this work came from the memory fragmentation problem in the Dali Object Storage Manager [BLR+97]. Dali is the research prototype for the DataBlitz main memory database system.

[2] Note that the algorithms presented are applicable whereever a set of objects have to be reorganized while being concurrently accessed, e.g. a stable heap [KW93]

1.1 Our contribution

We present a novel algorithm, the *Incremental Reorganization Algorithm* (IRA) which performs on-line reorganization with minimal interference to concurrently executing transactions. The crucial and complex part of IRA lies in how it efficiently determines the set of objects, $\mathcal{R}_O$, that reference an object O, with minimal interference to concurrent transactions. IRA uses a single fuzzy traversal (which uses only latches, and no locks) to determine an approximate $\mathcal{R}_O$ for all objects O being migrated; then for each O, one at a time, an exact $\mathcal{R}_O$ is found, locked, and the object is migrated. Thereby, very few locks are held at any time by IRA, thus minimizing interference with concurrent transactions. Moreover, The IRA is tolerant to failures in the sense that it tries to minimize the amount of wasted work.

For large databases, traversing the entire database in order to carry out a reorganization could be very expensive. IRA deals effectively with this problem by partitioning the database. IRA can be run on one partition at a time, thereby restricting traversal performed. Partitioning has been used in the past for garbage collection in object-oriented databases [YNY94, AFG95], and for reorganization of relational databases [Edi96].

The basic version of IRA requires transactions to follow *strict two-phase locking* (2PL), that is, all locks are held until the end of the transaction. This may be too restrictive in some high performance situations. We present two extensions to IRA, which improve concurrency further:

1. The first extension relaxes the strict two-phase locking requirement. Instead, transactions are only required to hold short duration locks on objects while accessing them.

2. The second extension does not require IRA to lock all objects in $\mathcal{R}_O$ simultaneously. Instead, it allows objects in $\mathcal{R}_O$ to be locked one at a time, while holding a lock on O, the object being migrated. Thus, at most two locks are held at any point in time by this extension.

We have implemented IRA and its extensions in Brahmā, a storage manager developed at IIT Bombay. We compare the performance of IRA to that of a system not running any reorganization. Our experiments demonstrate that for a wide range of workloads, the response times and throughput of the system while running IRA only degrade marginally. The maximum degradation in the average response time is around 5% while the maximum degradation in throughput is about 10%. We also compare IRA with a naive reorganization algorithm (which locks a significant portion of the database during reorganization). The average response times of transactions with the naive reorganization algorithm are significantly higher, and the throughput is correspondingly lower. More importantly, the variance in response times is several orders of magnitude higher with the naive algorithm, than with IRA. Thus, IRA is much better than the naive algorithm in ensuring that transactions are not adversely impacted by reorganization.

The rest of the paper is organized as follows: We outline our system model and assumptions in Section 2. Section 3 discusses the IRA algorithm in detail. The extensions and optimizations to IRA are discussed in Section 4. In the same section, we also discuss the failure handling and garbage collection aspects of our algorithm. We present the results of our performance evaluation in Section 5. We survey related work in Section 6 and present our conclusions and explore future directions for research in Section 7.

2 System Model

In this section, we describe the system model on which our on-line reorganization algorithm is based. The system model is very similar to the one used by Amsaleg et. al. [AFG95] and Roy et. al. [ARS+97] in their garbage collection work.

In our model, the objects in the database form a directed graph called the *object graph*. The nodes of the graph are the objects in the database, and an edge $R \rightarrow O$ exists in the graph if and only if R contains a reference to O. We assume that all references are physical. We use the term *reference* to mean the object identifier of an object, as well as to refer to an edge in the object graph, i.e., a reference from some object R to an object O. The intended usage will be clear from the context.

We shall refer to the objects R that reference an object O as the *parents* of O, and to O as a *child* of R. In our model, there exists a special object called the *persistent root*[3]. All objects in the object graph that are reachable either from the persistent root, or from an object whose reference is in the local memory of an active transaction are *live* objects; the rest of the objects in the database are not reachable (i.e, they are garbage). To traverse the entire graph, one can start at the persistent root, and follow references from one object to another.

We assume that the database is divided into units called *partitions*. We also assume that given an object identifier, we can inexpensively find the partition to which the object belongs[4]. The idea of partitioning has also been used by on-line garbage collection algorithms [YNY94, AFG95] and reorganization algorithms in relational databases [Edi96], to focus the problem on small units of the database.

The goal of partitioning is to be able to reorganize one partition at a time, and in particular, to be able to avoid traversing the entire database in order to find parents. Specifically, we wish to traverse only the objects in a partition, yet find all parents of objects that are to be migrated.

In order to do so, each partition $\mathcal{P}$ contains an *External Reference Table* (ERT), which stores all references $R \rightarrow O$ such that O belongs to $\mathcal{P}$ and R does not belong to $\mathcal{P}$. Thus, the ERT for partition $\mathcal{P}$ stores back pointers for references that come into $\mathcal{P}$ from other partitions. Objects O belonging to $\mathcal{P}$ that are noted in the ERT are called the *referenced objects of the ERT*. For simplicity, we assume that the persistent root is in a separate partition of its own, so that references from the persistent root to an object in any partition, is in the corresponding partition's ERT. We postpone for now the issue of how the ERT is maintained, and return to it in Section 3.3.

In this paper, for concreteness, we focus on the following specific reorganization problem: Given a partition $\mathcal{P}$, migrate all the objects in $\mathcal{P}$ to their specified new locations. This does not compromise the generality of our solutions; they can easily be extended if i) objects from multiple partitions have to be migrated and/or ii) only certain specific objects in the partition need to be migrated. We do not consider the problems of when to reorganize, which partition to reorganize and where the objects of the partition should be migrated. This is an orthogonal problem and the driving operation (e.g., compaction, clustering) makes these decisions.

We assume that transactions follow strict 2PL, i.e., all locks are held until the transaction commits or aborts (the algorithm is extended to relax this assumption in Section 4.1)[5]. A transaction can obtain a reference to an object only by following a sequence of references from the persistent root, unless it created the object. Once a transaction has locked an object O (in the appropriate mode), it can i) copy into its local memory any reference out of O, ii) delete a reference out of O and iii) insert a reference into O (i.e., store into O a reference to some object), copying it from the transaction's local memory. In all of the above, the transaction is not required to hold a lock on the referenced object.

For clarity of presentation we assume that objects are not created in the partition being reorganized after our reorganization algorithm starts execution[6].

We assume that the transactions follow the Write Ahead Logging Protocol **WAL**, i.e., they log the undo value before actually performing an update, but the redo value may be logged anytime before the lock on the object in question is released.

3 The Incremental Reorganization Algorithm (IRA)

In this section, we describe our *Incremental Reorganization Algorithm* (IRA), for reorganizing a partition. Before delving into the details of IRA, we outline a simple off-line algorithm for reorganizing a partition which assumes that the database is quiescent.

3.1 Reorganizing A Quiescent Database

We first consider how to reorganize a quiescent database, i.e., one on which no transactions are executing concurrently with reorganization. Reorganizing a partition $\mathcal{P}$ involves migrating each object O that belongs to $\mathcal{P}$. The basic steps in migrating an object O are i) find the parents of O, ii) move O to the new location, and iii) update the references to O in the parents of O.

In non-partitioned databases, finding the parents of an object requires a traversal of the object graph starting from the persistent root. However, in the case of partitioned databases, we do not have to traverse the entire graph; rather, we only traverse objects in the partition $\mathcal{P}$ that is being reorganized.

Traversal starts from all the objects that are referenced by objects external to the partition — these are exactly those objects referenced in the External Reference Table (ERT) of the partition. Whenever we traverse an edge from R to O, we add R to the list of parents of O. In addition to the parents found by traversing edges within the partition, we must add all parents from other partitions; these can be found in the ERT of partition $\mathcal{P}$.

[3]For ease of presentation, we assume there exists only one persistent root. Our algorithms can handle multiple persistent roots also.

[4]For example, the partition could be inferred from a fixed number of left most bits of the object identifier or some other hash function on the object identifier.

[5]We do not require locks to be held for transaction duration on schema level objects like index structures and collections. Updates to references, due to object migration, within these objects can be handled similar to relational databases.

[6]Our algorithms work correctly even if this assumption does not hold except it will not migrate objects created after the reorganization process starts execution. We outline how to extend our algorithm to migrate all objects created until some point of time after the reorganization process begins execution in [LRSS99]. Obviously objects created after the reorganization process completes can not be migrated.

Rather than performing a traversal of the partition once for each object, a single traversal is used to find the parents of all objects being migrated. As we perform the traversal, we construct multiple parent lists, one for each object we encounter in the course of traversal.

The assumption of the database being quiescent is important for the above algorithm, since concurrently executing transactions may update the object graph while traversal is going on. This could lead to the traversal missing some objects, or finding edges that get deleted later.

3.2 Outline of the IRA Algorithm

The above solution based on quiescing all database activity is too stringent for many applications. In contrast, the IRA algorithm allows transactions to execute on the partition all times during reorganization.

```
Incremental_Reorganization(𝒫) {
    (Objects, Parent_Lists) =
            Find_Objects_And_Approx_Parents(𝒫)
    For (each object O_old in the set Objects) do
        Find_Exact_Parents(O_old, Parent_Lists)
        Move_Object_And_Update_References(O_old,
            Parent_Lists)
}
```

Figure 1: Incremental Reorganization Algorithm

We now outline how the IRA algorithm is able to allow transactions to execute on the partition being reorganized. Figure 1 outlines the top level idea underlying IRA. As can be seen from Figure 1, the algorithm consists of two broad steps. The first step, implemented by the function Find_Objects_And_Approx_Parents finds the set of objects in the partition, and an approximation of the set of parents of these objects, by performing a fuzzy traversal of the partition. The objects, and the set of their corresponding parent lists are returned by the above function.

The fuzzy traversal does not obtain locks on the objects being traversed; instead, only a short term latch is obtained on the object for the duration of examining the references out of the object. The reason the traversal obtains only an approximation of the set of parents of an object is that parents of an object are constantly changing since other transactions execute concurrently. This step is explained in detail in Section 3.4.

The second step of IRA iterates over the set of objects discovered in the first step, and for each object, i) finds and locks the exact parents and then, ii) moves the object to its new location and updates references to the object. The second step is explained in detail in Section 3.5.

To help find the exact set of parents of an object, we collect all pointer inserts and deletes since the reorganization process started, in a data structure called the Temporary Reference Table (TRT). The TRT structure and its maintenance are described in Section 3.3.

3.3 Temporary Reference Table (TRT)

The *Temporary Reference Table* (TRT) of a partition $\mathcal{P}$, is a transient data structure, in which the deletion and addition of a reference to an object O in $\mathcal{P}$ are logged. The TRT structure is similar to the TRT used in [AFG95, ARS+97].

The TRT contains tuples of the form $(O, R, tid, action)$, where R is the referencer (parent) from which a reference to object O has been deleted or added by transaction tid; and *action* denotes whether the reference was inserted or deleted. We will call O above as the *referenced object* of the above tuple in the TRT, and the set of all such objects as the *referenced objects of the TRT*. A pointer delete must be noted in the TRT *before* the pointer is actually deleted by the transaction. Pointer inserts can be noted after the actual operation is done, but they should be made before the lock on the object in question is released.

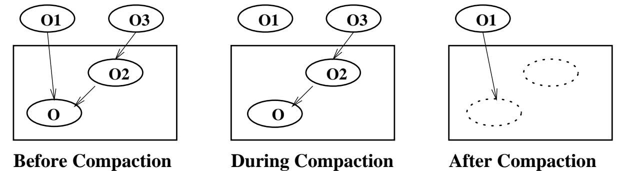

Figure 2: Motivating pointer delete logging in TRT

A simple mechanism to maintain the TRT and the ERT, as pointers are updated, is to process the system logs (as in [AFG95, ARS+97]) by a separate process called log analyzer[7] as soon as they are handed over to the logging subsystem. The log analyzer updates the TRT/ERT if the update log it is processing has caused a reference pointer to be inserted or deleted. Updates to the TRT itself are not logged, while updates to the ERT can be logged as in [AFG95]. Alternately, if the logging overheads for the ERT are perceived to be excessive, one can choose not to log updates to the ERT; however, in this case, we would then have to reconstruct the ERT at restart recovery.

We will now motivate the need for recording each of the above actions based on the high level description of IRA presented in Figure 1 and via the following examples:

Pointer Deletes: In the absence of the TRT, the following scenario is possible: Before IRA begins, a transaction T deletes a pointer from object $O1$ to an object O, but retains the reference to O in its local memory. When IRA runs, it would not find $O1$ to be a parent of O (and not try to lock 01 either, as a result).

After IRA migrates O, T may insert back the reference (either explicitly or due to an abort of T) to O; however, the reference would still point to the old location of O, which is now garbage. This scenario is illustrated in Figure 2. The TRT helps handle such cases by recording that a reference to O has been cut; IRA will wait for T (by attempting to lock $O1$ which is found by consulting the TRT) to complete before migrating O.

Another reason for logging pointer deletes, is to ensure that the fuzzy traversal does not miss out on some live objects in the partition. For example, if the only reference to O is deleted by a transaction T, O (and some of its descendants) may never be encountered during the fuzzy traversal. If T inserts a reference to O back after the traversal, IRA may not migrate O, although O is a valid object of the partition.

To fix this problem, IRA additionally performs a traversal from all objects in the TRT to which a reference has been deleted, as we will see. Thus, O (and all of its de-

[7]The TRT and ERT can also be maintained by other mechanisms like modifying the functions that perform pointer updates etc. For purposes of isolating this function, and to demonstrate that this kind of analysis can be added very easily to an existing system without disturbing existing user code, we have chosen to introduce a separate process for the TRT/ERT maintenance. The actual mechanism for maintaining the TRT/ERT is of no consequence to our algorithms.

scendants) will be encountered during the traversal, and migrated.

Pointer Inserts: As we mentioned earlier, IRA performs a fuzzy traversal and therefore may not encounter some pointer inserts that take place while IRA is in progress. However, these pointer inserts create new parents. Before migrating an object O, IRA consults the TRT to check if O has any new parents that IRA did not encounter during the traversal.

3.4 Finding Objects and their Approximate Set of Parents

As the first step towards reorganization of objects in a partition, we identify the set of all live objects in the partition, and for each object we identify an approximate set of its parents (3). To do so we perform a fuzzy traversal of objects in the partition, starting from objects referenced from the ERT.

```
Find_Objects_And_Approx_Parents(𝒫) {
/* Find the set of objects in the partition and their
approximate set of parents */
L1:  (Traversed_objects, Parent_lists) =
       Fuzzy_Traversal(referenced objects in ERT of 𝒫)
L2:  While (∃ a referenced object O in the TRT
     that is not in Traversed_objects)
           (Traversed_objects, Parent_lists) =
             Fuzzy_Traversal({O})
     return (Traversed_Objects, Parent_Lists)
}
```

Figure 3: Find Objects and Approximate Parents

The fuzzy traversal of the object graph is performed by Algorithm Fuzzy_Traversal, the pseudo code for which is not explicitly shown. Algorithm Fuzzy_Traversal starts traversal from the set of objects passed to it as its first argument and restricts the traversal to objects of the partition being reorganized. It adds the new set of objects encountered in a particular call to Traversed_Objects and adds the set of parents of these objects encountered in a particular call to Parent_Lists.

During the traversal, locks are not acquired on the objects encountered; instead, a latch is obtained to ensure physical consistency of the object while it is being read. The latch is released after the object has been read and all references out of the object have been noted. Thereby, the traversal is fuzzy, and does not return a transaction consistent view of the object graph within the partition. Note that even though not explicitly shown in the algorithm, latches on the shared data structures, ERT and TRT, need to be obtained whenever they are accessed. For clarity of presentation, we have omitted the actual latching details but note that the latch on TRT and ERT is not held while the Fuzzy_Traversal procedure executes.

The initial starting points for the traversal are the objects in ERT. The loop at line L2 is required to guarantee that no object in the partition is missed during the traversal. We will illustrate this with an example. Suppose the only reference to an object O is from R and this reference is cut before R is encountered by the Fuzzy_Traversal algorithm. This would result in O not being visited by the traversal and therefore not being recognised as a live object. Clearly, the transaction that cut the reference to O could reinsert it.

The following lemma states that all live objects are encountered by Find_Objects_And_Approx_Parents. This also enables IRA to detect and delete garbage objects (objects that are not live) simultaneously with reorganization. We will explore this connection in detail in Section 6.

Lemma 3.1 *When Algorithm Find_Objects_And_Approx_Parents completes, all live objects in the partition $\mathcal{P}$ are in Traversed_Objects.*

See [LRSS99] for the proof; due to space constraints, we omit it here.

An alternative to traversal from the ERT is to use object allocation information to find all objects in the partition, and visit all of them during traversal; doing so would not enable us to detect garbage objects, but would be otherwise the same.

3.5 Finding the Exact Set of Parents and Migrating an Object

We now explain the second step of the incremental reorganization algorithm. In this step, for each live object in the partition, we obtain the exact set of parents and then migrate the object. As we mentioned before, the set of parents of an object identified by the fuzzy traversal need not be exact. Function Find_Exact_Parents (pseudo code presented in Figure 4) makes this exact.

```
Find_Exact_Parents(O_old, Parent_Lists) {
S1:  Get Write Locks on all the objects in the parent
     list of O_old
     For each object R in the parent list of O_old
           If R is not a parent of O_old
                 Unlock R, and remove R from
                 parent list of O_old
S2:  While (∃ a tuple t in the TRT which has O_old
     as the referenced object)
           Write lock the parent object R of O_old in t
           Delete t from TRT
           If R is a parent of O_old
                 Add R to the parent list of O_old
           else Unlock R
}
```

Figure 4: Find Exact Parents

Find_Exact_Parents first obtains locks on the approximate parents of O_{old}, identified by the fuzzy traversal. If an object R is not a parent of O_{old} any longer (the reference was deleted after R was encountered during the fuzzy traversal), then R can be unlocked and removed from the parent list of O_{old}. Find_Exact_Parent next checks the TRT for the existence of a tuple containing O_{old} as the referenced object. If a tuple exists, then a reference to O_{old} from an object R has either been added or deleted. If a reference from R to O_{old} has been deleted, the transaction that deleted the reference has completed when IRA obtains a lock on R (by strict 2PL). Therefore, that transaction can no longer introduce a reference to O_{old} (any references introduced by it already will be in the TRT). If a reference from R to O_{old} has been added and R still contains that reference after a lock on R is obtained, then R is added to the parent list of O_{old}.

The while loop in Find_Exact_Parents terminates when there is no tuple in the TRT that contains O_{old} as the referenced object. The following two lemmas together

guarantee that all parents of O_{old} have been locked and there is no fear of a pointer to O_{old} reappearing in the database after O_{old} has been migrated.

Note that there is no need to obtain a lock on O_{old} itself, since the only way to access it is via a parent, and due to the strict 2PL requirement, no transaction can have a lock on O_{old} once all its parents are locked.

Lemma 3.2 *All live objects that have a reference to O_{old} at the time Find_Exact_Parents completes are locked by IRA.*

Proof: We provide an proof sketch here. A more formal proof can be found in [LRSS99]. Let t be the time instant at which Find_Exact_Parents completes. Let us assume that there exists a live object R containing a reference to O_{old} at time t that has not been locked by IRA. We will consider two cases:

Case 1: *The reference was added before the reorganization algorithm started*
By Lemma 3.1, R would have been encountered by the fuzzy traversal. Therefore, the reference also would have been encountered at the time R was encountered and therefore R is in the parent list of O_{old} at t. Therefore, R would have been locked by IRA at statement S1 of Find_Exact_Parents – a contradiction.

Case 2: *The reference was added after the reorganization algorithm started*
Let T be the transaction that added the reference. If T has completed at t, then the insertion would be logged in the TRT and the loop at S2 would have caught this and locked R. Therefore, T has still not completed at t. Since, we do not allow creation of objects after the reorganization process starts, T should have obtained a reference to O_{old} from some other object. Let R' be the object in which T first found a reference to O_{old}. If the reference from R' to O_{old} is not present at time t, then T must have deleted it, and by WAL, this deletion must be in TRT at time t. Therefore, IRA must have obtained a lock on R' in the loop at S2, which is impossible since T has a lock on R' at t. Therefore, the reference from R' to O_{old} is present at time t. Without loss of generality, we can assume IRA has a lock on R' (otherwise we can keep repeating the proof of this lemma for the reference from R' to O_{old} and this will push back the time of addition of the reference being considered in the proof until the time of addition of the reference satisfies Case 1). However, T holds a lock on R' at t – a contradiction.

Lemma 3.3 *There does not exist an active transaction that has a reference to O_{old} in its local memory at the time Find_Exact_Parents completes.*

The proof is similar to the proof of Lemma 3.2; see [LRSS99] for details.

It now follows that no transaction in the future can obtain a reference to O_{old} and so O_{old} can be safely moved. The move is performed by Move_Object_And_Update_Ref, the pseudo code for which is presented in Figure 5. This is essentially a bookkeeping function that actually effects the migration of O_{old} and ensures all references to the object at the old location refer to the new location and that all the ERTs are consistent with the migration. O_{new} is made visible to other transactions after this function completes since the locks on the parents of O_{old} are released at the end. We treat the migration of each object as a transaction (see section 4.3). In other words, the calls to Find_Exact_Parents and Move_Object_and_Update_Refs for a particular object

```
Move_Object_And_Update_Refs(O_old, Parent Lists) {
    Copy O_old to the new location, say O_new
    For each parent R in the parent list of O_old
        Change the reference in R to point to O_new
        Update ERTs of the partitions where O_old
        and O_new reside to reflect the change
    For each child C of O_old that is in the partition
    being reorganized
        If C is not yet migrated
            Replace O_old by O_new in the parent list of C
    For each child C of O_new
        Update the ERT of the partition corresponding
        to C to reflect the migration
    delete O_old
    Unlock all the parents of O_old (O_new)
}
```

Figure 5: Move Object and Update References

O_{old} in Figure 1 are together executed in the context of a transaction. As a result, system failures will not undo the migration of objects, if the transaction in whose context the object was migrated has completed. The migration of an object which was in progress at the time of failure (if any) will be undone. The reorganization process has to be started afresh to migrate the objects yet to be migrated. Alternatively, the reorganization process can log information about its execution state, to ensure that it does not have to be started afresh in case of a system failure. The impact of failures is further explored in section 4.4.

4 Extensions and Optimizations

In this section, we consider two important extensions to IRA. The first does away with the assumption that transactions follow strict 2PL. The second reduces the number of concurrent locks held by IRA. We then describe how to aggressively reclaim space occupied by the TRT and limit the amount of logging to the TRT.

4.1 Relaxing Strict 2PL Assumption

In this section, we show how the assumption that transactions follow strict 2PL can be relaxed. We assume, however, that before accessing an object, a short duration lock is obtained; the lock may be released after the object has been accessed.

For the reorganization process to work correctly when transactions do not follow strict 2PL, we augment the lock manager to keep track of which active transactions had acquired short duration locks on which objects. Whenever the IRA locks an object, it must additionally wait for all active transactions that have ever acquired a lock on this object to complete. Thus, the IRA waits for transactions that may have copied a reference into its local memory but may not currently hold a lock on the source of the reference. This results in transactions behaving as though they were following strict 2PL with respect to the reorganization process.

4.2 Reducing the Number of Concurrent Locks

The algorithm proposed in the previous section requires all parents of an object to be locked before it can be relocated. However, for objects with a large number of parents, this could prove to be restrictive since a substantial portion of the database may end up getting locked.

In this section, we show how to reduce the number of concurrent locks held by IRA. Rather than obtaining a lock on all the parents of an object, and then migrating the object and updating references in the parents, we lock the object being migrated (in both the old and the new locations) and then lock the parents one at a time, releasing the lock on a parent before obtaining a lock on the next parent. The reference in the parent is updated to the new location of the object while the parent is locked. As a result, each parent update is now done within the context of a transaction as opposed to each object migration in a transaction as described in section 3.5 (also see section 4.3). Please refer to [LRSS99] for the pseudo code for this extension.

No transaction can obtain a lock on the object being migrated since it is locked by the IRA while the object is being migrated. Transactions can however copy references to both O_{new} and O_{old} into other objects. We can ignore new references to O_{new} since they are correct after relocation. New references to O_{old} will be detected using the TRT as described earlier. Thus, it is correct to obtain locks on one parent at a time. This extension is a very powerful optimization since locks are held on at most two distinct objects at any point of time.

However, the algorithm can lead to the following situation: For an object O being migrated, there may exist a parent R which references O_{old} and another parent R' which references O_{new}. This has the following repercussions:

- In the event of a system failure, after restart recovery, the database may have two different objects, one pointing to O_{old}, and the other pointing to O_{new}. In this case, both O_{old} and O_{new} need to be locked before allowing transactions to start execution. The Reorganization process can then be restarted.
- The references to O out of R and R' do not match. Therefore, any transaction that attempts to compare references will obtain an erroneous result. Thus, this optimization is valid only if either (1) transactions do not compare references without obtaining locks on the referenced objects, or (2) the comparison operation does an additional check to see if the two referenced objects are old and new versions of an object being migrated – if so, the two references are considered to be equal.

Note that this optimization can be used in conjunction with the extension of Section 4.2.

4.3 Transaction context for Object Migration

In section 3.5, we noted that each object migration is done within the context of a transaction. In the extended version of the algorithm in section 4.2, each parent update is done in the context of a transaction. As mentioned earlier, this will ensure that work completed once will not be lost on a failure and will not have to be repeated upon recovery. To keep the descriptions of the algorithms simple, we used a seperate transaction for each object (or each parent in the extension). In practice, this could impose a high logging and IO overhead. To address this problem, we note that it is not required for correctness that each operation be a seperate transaction. Multiple object migrations can be grouped into a transaction (similarly, multiple parent updates can be grouped in the extension) to reduce the logging overhead. The trade-off here is between the size of the transaction and the amount of work that may need to be repeated after a failure. In the next section, we look at some other optimizations related to failure handling.

4.4 Handling Failures

In this section, we consider the effect of failures on the ERT, and on the two steps of the IRA algorithm, in turn.

1. If the ERT is to be persistent, the updates made by the log analyzer to the ERT should also be logged. This logging is performed as though these updates were made by the original transaction whose log is being analyzed. This ensures that aborts of transactions and restart recovery do not have to do anything special to keep the ERT consistent. This was the approach taken in [AFG95].

   Alternatively, if the logging overheads for the ERT are perceived to be excessive, one can choose not to log updates to ERT; however, in this case, we would then have to reconstruct ERT at restart recovery, which requires a complete scan of the database. An intermediate solution is to checkpoint ERT periodically and use the logs for pointer deletes and inserts during restart recovery to bring the ERT up to date.

2. Algorithm Find_Objects_And_Approx_Parents which is the first step of IRA does not obtain any locks. Therefore, it can never be involved in a deadlock. A system failure during Find_Objects_And_Approx_Parents would however result in the loss of the work performed until the failure.

   A simple solution to system failures during Find_Objects_And_Approx_Parents is to restart the IRA algorithm on restart recovery. However, if the loss of work is unacceptable, the data structures Traversed_Objects and Parent Lists can be checkpointed periodically. In the event of a failure, the TRT is reconstructed on the basis of the logs generated after the IRA started.

   Optionally, the TRT could also be checkpointed and then only the logs after the checkpoint need to be considered during the TRT reconstruction. In any case, after the TRT is reconstructed, the last checkpoint of the data structures can then be used to reduce the work of Find_Objects_And_Approx_Parents (by not traversing parts of the graph which have already been traversed).

3. The second step of IRA is to invoke Find_Exact_Parents and Migrate_Object for each object in the partition – we perform this within a transaction. Therefore, once a call to these functions succeed for an object O, the migration of O is complete.

   Migrate_Object does not obtain any locks and can not be involved in a deadlock. Find_Exact_Parents has to be reinvoked if it fails due to a deadlock.

   After a system failure during the second step, the objects that have not yet been migrated need to be migrated. If Traversed_Objects and Parents Lists are checkpointed after the completion of the first step, then the TRT can be reconstructed after a system failure by performing a scan of the system logs and the second step (to migrate remaining objects) can be started right away after recovery from failure. If the work done in the first step is lost during a system failure, IRA is started afresh from the beginning for the objects yet to be migrated.

4.5 Minimizing Space and Time Overhead of TRT

In this section, we describe methods to reduce the time and space overheads of the TRT. First, note that the TRT

on a partition is required only if a reorganization process is in progress and does not exist otherwise. Once the reorganization process starts, the log analyzer starts noting relevant updates in the TRT. The reorganization process waits for all transactions that are active at the time it started, to complete, before starting the fuzzy traversal. This ensures that all relevant updates are indeed present in the TRT.

If transactions follow strict 2PL, the tuples corresponding to pointer deletes in the TRT can be deleted as soon as the transaction that logged them completes (aborts or commits). The main reason pointer deletes are logged is to ensure that any reinsertion of the reference by the transaction that deleted the pointer is seen by the reorganization process. Since insertions of pointers are logged separately in the TRT and pointers cannot be cached (in local memory) across transaction boundaries, this deletion of the tuple from the TRT does not compromise correctness. We assume here that if the abort of a transaction reintroduced a deleted reference, it is treated as an insertion of a reference, so the insertion remains in the TRT. Moreover, when a transaction that deleted a reference from R to O commits, we can also delete any tuple (if it exists) in the TRT that corresponds to the insertion of the reference from R to O.

Note that if transactions do not follow strict 2PL, a reference deleted by a transaction T may have been seen by another transaction T' which may reinsert the reference after T commits. Thus, for the non-2PL case, we do not allow the TRT tuples corresponding to deleted pointers to be purged after the transaction that deleted the pointer completes.

4.6 Relationship to Garbage Collection

One of the advantages of performing a traversal of the object graph is that the live objects of the partition can be detected. If we were only performing garbage collection, then a sweep through the partition would have identified the garbage objects. This is essentially the partitioned Mark and Sweep algorithm [AFG95]. The other option for garbage collection is to migrate all the live objects of a partition to a new partition and reclaim the entire space in the old partition. This is essentially the partitioned copying collector algorithm [YNY94]. However, the copying collector algorithm of [YNY94] assumes the object references are logical. Migrating objects when references are physical is hard – the focus of this paper is to attack this problem. However, as a side effect, we also have a partitioned copying collector algorithm even if the references are physical. The authors in [YNY94], in fact, advocated this algorithm over Mark and Sweep due to the ability of the algorithm to recluster the database. Thus, a system that implements our reorganization utility does not require a separate garbage collection utility.

5 Performance Evaluation

In this section, we investigate the impact of IRA on the performance of concurrent transactions. We consider the following: (a) a system running the basic version of IRA (without the extensions described in Section 4), (b) a system that is not running a reorganization utility, henceforth called NR, for No Reorganization, and (c) a system that runs a reorganization utility which quiesces the entire partition being reorganized (described in Section 5.1). The reorganization utilities were added to Brahmā, a storage manager developed at IIT Bombay. Brahmā provides support for the strict 2PL protocol and supports WAL through an implementation of ARIES. Brahmā also supports extendible hash indices which were used to implement the TRT and the ERT. A lock timeout mechanism was used to handle deadlocks and was set to one second throughout the experiments. Our experiments were conducted on a standalone 167 Mhz Sun UltraSparc-1 machine running Solaris 2.6 and equipped with 128 MB of RAM.

5.1 Partition Quiesce Reorganization (PQR)

We now outline the Partition Quiesce Reorganization Algorithm (PQR), which quiesces the partition being reorganized before performing the reorganization. By quiescing a partition, we mean that no reference to an object in the partition can be added or deleted. This is essentially a scaled down version of the off-line algorithm of Section 3.1.

However, while it is trivial to ensure no transaction is active in the off-line algorithm, it is bit more complex to ensure the partition being reorganized is quiescent. To quiesce a partition, we need to locks all objects not in the partition, that have a reference to an object in the partition. This ensures that no transaction can obtain a reference to an object in the partition.

Like in the IRA a TRT is maintained to detect insertion of new references while other parents are being locked. The following pseudo code outlines how a partition is quiesced. Once a partition is quiesced, reorganization is straightforward.

```
Quiesce_Partition(P)
  while (∃ a parent object R in ERT which is not locked)
     Lock R
  while (∃ a parent object R in TRT which is not locked)
     Lock R
```

Note that as in the case for IRA there is no need to lock the objects in the partition, since any transaction that accesses these objects must have come in through some external parent (possibly the persistent root). The external parent would be locked above, so with strict 2PL no transaction could be accessing any object in the partition.

5.2 Workload

We now describe the workload we consider in our experiments. This includes how the object graph is structured, and the access pattern of transactions. Table 1 shows the parameters used in the experiments.

Parameter	Meaning	Default
NUMPARTITIONS	partitions in the database	10
NUMOBJS	objects per partition	4080
MPL	multi programming level	30
OPSPERTRANS	length of random walk per transaction	8
UPDATEPROB	probability of exclusive access	0.5
GLUEFACTOR	fraction of inter-partition references	0.05

Table 1: Parameters of the implementation

Object Graph Structure The object database is made of NUMPARTITIONS number of partitions, each containing NUMOBJS number of objects. In each partition, the objects are organized into clusters, where each cluster is a tree with 85 objects. The roots of these clusters are treated as persistent roots. One edge from each node in the cluster refers to a node in another cluster C; C is chosen to be

in another partition with probability GLUEFACTOR; thus controlling the number of inter-partition references.

Transaction Access Pattern The multi programming level (MPL) determines the number of concurrent transactions in the system at any given time. The multi programming level is fixed by spawning MPL threads that submit transactions to the system. When a transaction submitted by a thread completes, the thread submits the next transaction.

A typical transaction performs a random walk through the object graph. All the transactions in a particular thread start their random walk in a specific "home" partition and the threads are uniformly assigned to all the partitions. For the transactions in a particular thread, the starting points of the random walks are chosen randomly from among the persistent roots in the partition assigned to that thread.

The random walk chooses the next object to be accessed randomly from the references out of the current object. The number of objects accessed during the random walk is given by the OPSPERTRANS parameter which is fixed at 8. At each stage in the walk, the transaction may get a lock on the next object in shared or exclusive mode. This is determined by a parameter called the UPDATEPROBABILITY which represents the probability that an access performed by the transaction is an update access.

5.3 Results of Experiments

Previous studies [YNY94, AFG95, ARS+97] have studied the I/O overhead of partition traversal and found it to be reasonable. Therefore, we do not address this issue in our experiments. They have also studied the impact on normal processing of maintaining the TRT and ERT (and shown it to be reasonable) and therefore we do not address that issue either in this section.

In all our experiments, the database was kept memory resident for two reasons: i) In many high performance situations like telecommunications applications which require physical references, the database is indeed memory resident and ii) The goal of the experiments were to stress on the concurrency and data contention aspects of our algorithms, and not the I/O aspects as we mentioned above. We plan to investigate the performance in a disk-based setting in the future.

We evaluated the algorithms based mainly on the two performance metrics: i) Throughput and ii) Average Response Time. Transactions were run until the reorganization operation completed in the case of IRA and the PQR algorithm. In all cases, IRA takes longer to complete the reorganization than the PQR algorithm. Measuring the throughput and the response time of the transactions while reorganization is being performed is a direct measure of the impact of these algorithms on concurrent transactions. In the case of the system where there was no compaction, we ran 10,000 transactions in each thread. Since our experiments were on a real system, all times are measured as the wall clock elapsed time.

In the following sections, we report the results of the experiments in which we varied one parameter while others were fixed at the default values shown in Table 1. All the times are in milliseconds and throughput is measured in transactions per second (tps) unless otherwise mentioned.

5.3.1 MPL

In this section, we evaluate the performance of the algorithms as the MPL is varied. The purpose of these experiments is to examine in detail the difference in throughput for a range of system load.

Figure 6 shows the throughput of the algorithms as the MPL is varied. The NR system has consistently the best throughput as would be expected. The throughput of IRA is very close to the NR system for all the MPL values, while the throughput of the PQR algorithm is significantly lower.

Note that the throughput of the NR system peaks around an MPL of 5 since resource contention set in around that value. Recall that the entire database is memory resident; therefore CPU gets saturated very soon. The reason the throughput does not peak at an MPL of one is that logs have to be flushed to disk at commit time; therefore, there is some CPU I/O parallelism to be exploited. The throughput of IRA also peaks around the same value for similar reasons.

The throughput of the PQR algorithm peaks much later, around an MPL value of 30. The reason PQR algorithm peaks later is it causes severe data contention and underutilizes the resources at low MPL values. In fact, at some point of time during the reorganization, even at an MPL of 30, all threads get blocked by the reorganization process. To understand this, note that all the external parents of the objects in the partition are locked. Clearly, the threads that originate their walk in the partition being reorganized (there are three of them at an MPL of 30 since there are 10 partitions), cannot proceed since the persistent roots of this partition are considered to be external to the partition and therefore locked. Moreover, some transaction in the other threads may eventually need to lock one of the external parents (which could be in any partition). Thus, over a period of time some transaction of each thread ends up waiting for the reorganization process in the PQR algorithm.

The above effect will get amplified dramatically in a system with plenty of resources like a multi-processor environment. The throughput of NR and IRA would scale with additional resources as we increase the MPL while the throughput of PQR would stagnate due to the excessive data contention which essentially locks out all transactions from the system.

Figure 7 shows the average response times of the respective algorithms. The average response times reflect the throughput curves. We now analyze the response times of transactions, at an MPL of 30, where the PQR algorithm achieves its peak throughput, a bit more carefully.

	Throughput	Avg Resp. Time (msec)	Max Resp. Time (msec)	Std. Devn. of Resp. Times
NR	35.0	819	1503	127
IRA	33.7	861	1935	135
PQR	28.0	1030	100040	4113

Table 2: Analysis of Response Times

Table 2, in addition to the throughput and the average response time, shows the maximum[8] of the response times of the transactions and the standard deviation of the response times of the transactions. The standard deviation and the maximum of the response times of transactions in the NR system and the system with IRA are very close. This is a very desirable behavior of a utility, since this implies that concurrent transactions in effect do not see the utility. Predictability of response times is also very important in real time systems and IRA scores on this count too. In contrast, the PQR algorithm affects concurrent transactions severely,

[8] Though, we have shown only the maximum response time, the trend remains the same even when we consider the 10 highest response times or the average of the top 10 response times

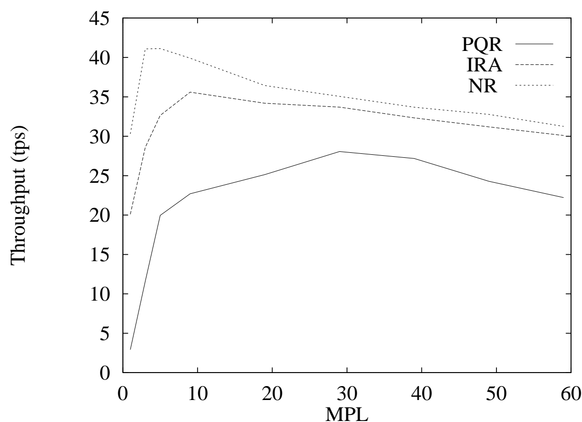

Figure 6: MPL scaleup - Throughput

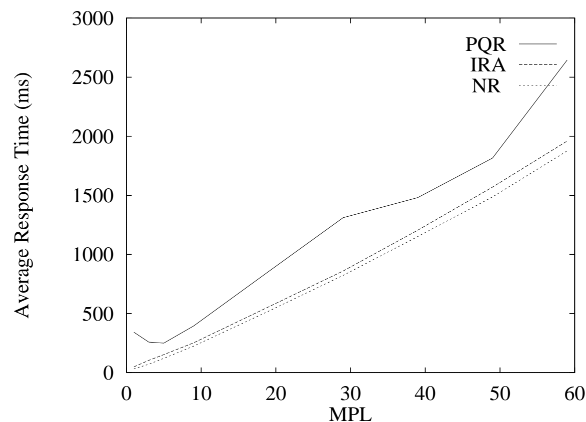

Figure 7: MPL scaleup - ART

and in fact, brings the system to a grinding halt, eventually, as explained above.

5.3.2 Number of Objects in a Partition

In this section, we compare the algorithms when the size of the partitions is scaled up. Figures 8 and 9 show the relative performance of the algorithms. As the partition size increases, the number of objects that need to be relocated and hence the number of parent pointers that need to be updated and the number of locks that need to be obtained increases. Because of this, the reorganization takes a longer time as partition size increases.

First, we observe from Figure 8 that the throughput is quite steady[9] both in the IRA case and the NR case, as the number of objects in the partition grows. On the other hand, the throughput of the PQR algorithm drops consistently as the number of objects in the partition is increased. In locking the entire partition, for the duration of the reorganization, which takes a longer time as partition size increases, PQR blocks transactions for a longer time. Thus, it monopolizes system resources for a longer time and also causes greater wasted work due to aborts (as a result of timeouts) of transactions.

Coming to the average response time, we can observe from Figure 9 that the increase in case of PQR is much more dramatic than that in case of IRA. This can be attributed again to the increased amount of time transactions have to wait for the reorganization process to complete.

5.3.3 Update Probability

Next, we studied the effects of varying the update probability. Figures 10 and 11 show the throughput and the average response time of transactions as the update probability is varied. In fact, this experiment is the final confirmation that the default values chosen by us were more than fair to the PQR algorithm. MPL values lower than 30 (running experiments beyond an MPL of 30 is pointless since we are examining areas of very high contention which is not reflective of normal processing), partition sizes larger than 4080 (with an average object size of 100 bytes a partition is around 400K which is very small), update probability smaller than 0.5 (which is much more normal) would only skew the results even more in favor of IRA.

[9]There is a very small variation of less than 2% which is within the experimental noise since we were measuring wall clock time on a real system

The PQR algorithm is relatively less affected by an increase in update probability since the data contention is severe even at low update probability. Therefore, an incremental increase in update probability has a much higher impact on IRA and NR. The performance of PQR algorithm still remains lower than IRA even for very high update probability values.

5.3.4 Other Experiments

We also examined the performance of the algorithms as the other performance metrics like the gluefactor, the transaction path length and the number of partitions were varied. Due to space constraints we skip the details of these experiments and refer the reader to the full version of the paper [LRSS99].

Finally, we repeated all our experiments while measuring throughput and response time of the PQR algorithm for the duration of IRA rather than just the duration of the PQR algorithm. The motivation for this study is the following observation: While it is true that the PQR algorithm affects concurrent transactions severely for the duration of reorganization, it brings back normalcy much faster. Thus, for the extra duration that IRA requires to complete reorganization, a system with PQR algorithm would have throughput as though there were no reorganization. Thus, a natural question is what is the loss in throughput of IRA compared to PQR if throughput is measured for the duration of IRA for both algorithms. We found the difference in throughputs never exceeded 3%. Note that the throughput of PQR can never exceed that of NR and since the difference between IRA and NR was never big in the first place, the difference between IRA and PQR could never have been big.

6 Related work

The only prior work [KW93] we know of for on-line reorganization of objects with physical references requires an action-quiescent state during which all objects that are in the memory of active transactions and persistent roots are copied into a new space. This could cause severe intereference to concurrent transactions. Apart from the above, their model is very different from ours and requires low level support from the hardware and the operating system. For example, they require, that they be able to change references in the registers and stacks of active transactions and trap certain pointer dereferences using memory protection. Finally, they use forwarding addresses, which may cause an extra I/O and require complicated

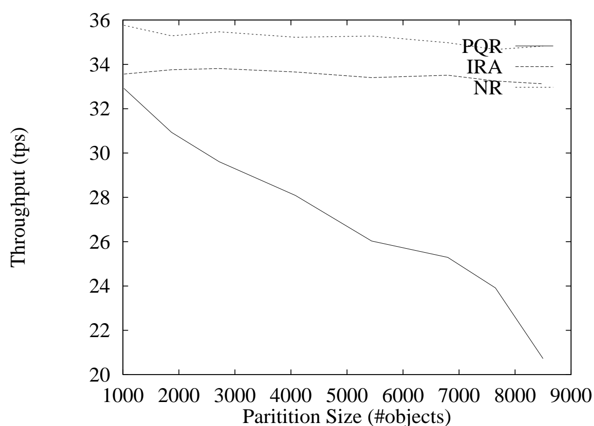

Figure 8: Partition size scaleup - Throughput

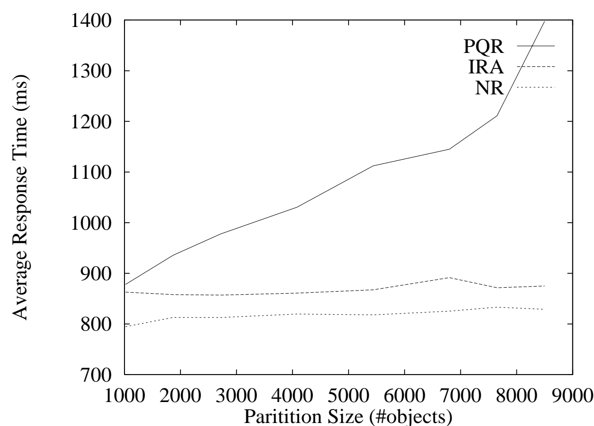

Figure 9: Partition size scaleup - ART

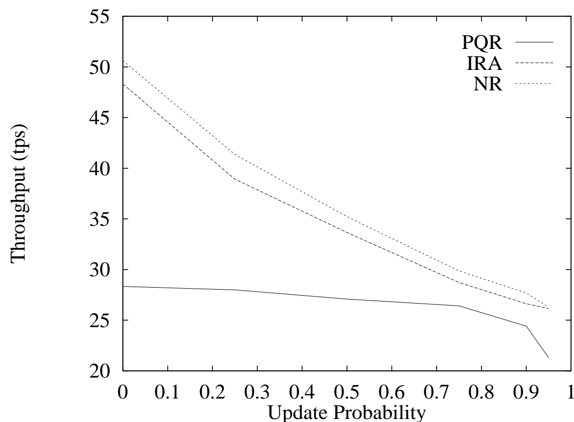

Figure 10: Update Probability - Throughput

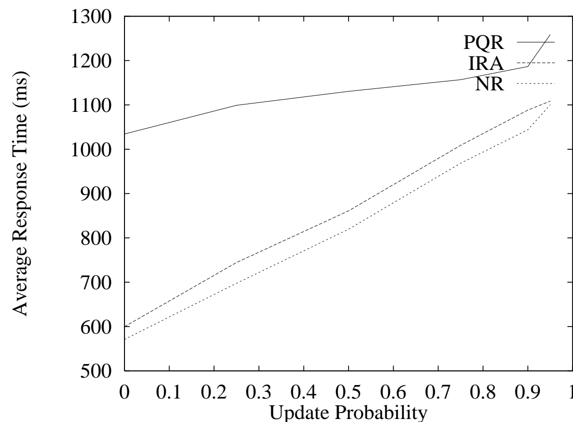

Figure 11: Update Probability - ART

recovery mechanisms to ensure consistency of the disk version of the database.

On-line reorganization is a well studied topic in relational databases [AON96, Omi96, SD92, ZS96a, ZS96b, ZS98]. In a relational database, references to a record are restricted to be from the indexes on the table. Thus, discovering the set of references to a record is a much simpler task in relational databases. Research into reorganization in relational databases has therefore concentrated on minimizing the number of locks being held and the amount of I/O necessary for reorganization.

The motivation for the problem of reorganization of object-oriented databases when references are physical arises from the fact that logical references impose an overhead on every access of the object. In [EGK95], three alternatives for the implementation of logical object identifiers are compared. The best technique based on direct mapping, often requires one extra I/O per object access which is unacceptable in situations where low response times are desired. Moreover, the very motivation for the work in [EGK95] is the inability to perform reorganization efficiently when references are physical which is exactly the focus of this paper. Finally, while it is true that if references are physical, quite a few object references have to be changed during the reorganization, it is important to note that reorganization is an infrequent occurrence, that can be performed at lean times, whereas, using a logical identifier would mean the price of the extra level of indirection is paid for *every* access.

There are several problems other than reorganization with an on-line flavor. On-line index construction algorithms [SC92b, MN92] build an index on a relation while the relation is being updated. The concurrent updates are collected in a side file and applied later. The entire relation is locked for a very short duration to enable the on-line operation to catch up. This idea was also extended to the execution of large queries [SC92a].

The problem most related to on-line reorganization in object-oriented databases is on-line garbage collection [YNY94, AFG95, ARS+97] in object-oriented databases. The system model is very similar in both the problems. There are two broad classes of garbage collection algorithms. The first consists of algorithms that migrate live objects elsewhere. The partitioned copying collector algorithm of [YNY94], where all the live objects of a partition are migrated out of the partition, is an example. These algorithms however assume object references are logical. Migrating objects when references are physical is much more difficult since it requires the references in all parents to be updated correctly. The focus of this paper is exactly on this problem. Since, our reorganization algorithm also detects all live objects in a partition, it can easily be augmented to copy all objects out of the partition, similar to the partitioned copying collector algorithm. However, our reorganization algorithm works correctly even when references are physical. The second class consists of algorithms that do not migrate live objects but perform garbage collection in place. Reference Counting and Mark and Sweep based algorithms belong to this class.

[AFG95] is a partitioned mark and sweep algorithm while [ARS+97] is a combination of reference counting and mark and sweep. These algorithms can handle physical references but do not perform any reorganization. Therefore, they do not perform the crucial step of finding and locking all parents of an object. In fact, our algorithm holds locks on atmost two distinct objects at any point of time and in contrast to [YNY94, AFG95, ARS+97], does not require transactions to follow strict 2PL. In summary, our algorithm can perform garbage collection and reorganization and yet allow references to be physical, an ability that to the best of our knowledge, no previous algorithm in literature possesses.

The notion of TRT has been used in previous work on garbage collection and reorganization in relational databases (called side files in this context). However, the space optimization measures we present in Section 4.5 are novel to this paper. The orthogonal issue of how to choose the partition size to minimize the overhead of the partition traversal has been addressed in [CWZ94].

7 Conclusions and Future Work

We have considered the problem of on-line reorganization in an object oriented database where object references are physical, and presented the IRA algorithm, and several variants of it to improve concurrency. One of the variants holds locks on only two distinct objects at any point of time, and does not require transactions to follow strict 2PL. The experiments we conducted confirmed that the IRA algorithm interferes very little with concurrently executing transactions. Thus, for database systems that employ physical object references for higher performance (e.g., main-memory database systems), IRA ensures that they do not pay a very high penalty during object reorganization.

In the future, we plan to address the issue of improving the I/O efficiency of the reorganization process. Even if the partition being reorganized fits in memory, the external parents of the objects in the partition may not. An object external to the partition being reorganized may have to be fetched multiple times as it may be the parent of multiple objects in the partition. A natural question that arises is in what order to we migrate objects so that the number of I/O's required is minimized. In a main memory database, the same order could be relevant since it may minimize the number of times locks have to be obtained on an external object. In the near future, we plan to carry out a detailed performance study of our algorithms in a disk-based setting.

References

[AFG95] L. Amsaleg, M. Franklin, and O. Gruber. Efficient incremental garbage collection for client-server object database systems. In *Proceedings of the 21st VLDB Conference*, September 1995.

[AON96] K. Achyutuni, E. Omiecinski, and S. Navathe. Two techniques for on-line index modification in shared nothing parallel databases. In *Proceedings of ACM SIGMOD Conference*, pages 125–136, Montreal, June 1996.

[ARS+97] S. Ashwin, P. Roy, S. Seshadri, A. Silberschatz, and S. Sudarshan. Garbage collection in object oriented databases using transactional cyclic reference counting. In *Proceedings of the 23rd VLDB Conference*, Athens, Greece, August 1997.

[BKKK87] J. Banerjee, W. Kim, H. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of ACM SIGMOD Conference*, pages 311–322, 1987.

[BLR+97] P. Bohannon, D. Lieuwen, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. The architecture of the dali storage manager. *Journal of Multi-Media Tools and Applications*, 4(2), March 1997.

[CWZ94] J.E. Cook, A.L. Wolf, and B.G. Zorn. Partition selection policies in object database garbage collection. In *Proceedings of ACM SIGMOD Conference*, pages 371–382, Minneapolis, USA, May 1994.

[Edi96] B. Salzberg (Special Issue Editor). Special issue on online reorganization. *IEEE Data Engineering Bulletin*, 19(2), June 1996.

[EGK95] A. Eickler, C. A. Gerlhof, and D. Kossman. A performance evaluation of oid mapping techniques. In *Proceedings of the 21st VLDB Conference*, September 1995.

[JLR+94] H.V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main-memory storage manager. In *Proceedings of the 20th VLDB Conference*, 1994.

[KW93] E.K. Kolodner and W. E. Weihl. Atomic incremental garbage collection and recovery of a large stable heap. In *Proceedings of ACM SIGMOD Conference*, pages 177–186, Washington, DC, May 1993.

[LRSS99] M.K. Lakhamraju, R. Rastogi, S. Seshadri, and S. Sudarshan. On-line reorganization of objects. In *Technical Report, Bell-labs*, February 1999.

[MN92] C. Mohan and I. Narang. Algorithms for creating very large tables without quiescing updates. In *Proceedings of ACM SIGMOD Conference*, pages 361–370, San Diego, USA, May 1992.

[NOPH92] S. Nettles, J. O'Toole, D. Pierce, and N. Haines. Replication-based incremental copying collection. In *International Workshop on Memory Management*, pages 357–364, St. Malo, France, September 1992.

[Omi96] E. Omiecinski. Concurrent file reorganization: Clustering, conversion and maintenance. *IEEE Data Engineering Bulletin*, 19(2), 1996.

[SC92a] V. Srinivasan and M. Carey. Compensation based on-line query processing. In *Proceedings of ACM SIGMOD Conference*, San Diego, CA, 1992.

[SC92b] V. Srinivasan and M. Carey. Performance of on-line index construction algorithms. In *3rd International Conferance on Extending Database Technology*, pages 292–309, Vienna, Austria, March 1992.

[SD92] B. Salzberg and A. Dimock. Principles of transaction-based on-line reorganization. In *Proceedings of 18th VLDB Conference*, pages 511–520, 1992.

[TN91] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In *Proceedings of the ACM SIGMOD Conference*, Denver, Colorado, May 1991.

[WMK94] Jr W.J. Mciver and R. King. Self-adaptive, on-line reclustering of complex object data. In *Proceedings of ACM SIGMOD Conference*, pages 407–418, Minneapolis, USA, May 1994.

[YNY94] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proceedings of the Data Engineering International Conference*, pages 120–133, February 1994.

[ZS96a] C. Zou and B. Salzberg. On-line reorganization of sparesely-populated B^+-trees. In *Proceedings of ACM SIGMOD Conference*, pages 115–124, Montreal, June 1996.

[ZS96b] C. Zou and B. Salzberg. Towards efficient online database reorganization. *IEEE Data Engineering Bulletin*, 19(2):33–40, June 1996.

[ZS98] C. Zou and B. Salzberg. Safely and efficiently updating references during on-line reorganization. In *Internation Conference on Very Large Databases*, New York, USA, August 1998.