Efficient Garbage Collection for Large Object-Oriented Databases

by

Tony C. Ng

B.S., University of Illinois, Urbana-Champaign (1994)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author ....
Department of Electrical Engineering and Computer Science
May, 1996

Certified by ....
Barbara H. Liskov
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Efficient Garbage Collection for

Large Object-Oriented Databases

by

Tony C. Ng

Submitted to the Department of Electrical Engineering and Computer Science
on May, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis presents the design of an efficient garbage collection scheme for large, persistent object-oriented databases in a client-server environment. The scheme uses a partitioned approach. A database is divided into disjoint partitions and each partition is collected independently. The scheme maintains transaction semantics and survives server crashes. It can be run concurrently with client applications. It also compacts objects in the partition being collected.

The garbage collector has been implemented in Thor, a distributed object-oriented database. It runs as a low-priority thread in the system. The collector interacts with the clients incrementally to obtain root information for collection. Client applications can continue to fetch objects from the database or commit transactions with no extra overhead while garbage collection is happening.

To allow partitions to be collected independently, the database server has to maintain a list of inter-partition object references for each partition. We call this list an inter-partition reference list (IRL). One major problem with any partitioned garbage collection scheme is the cost associated with the updates of IRLs, which have to be maintained persistently because they are too expensive to recompute after server crashes. We describe a new technique that reduces the number of I/O operations required to update the IRLs. IRL update operations are logged and deferred. The number of I/Os are reduced by re-ordering and processing the IRL updates in batches at some later time.

Keywords: garbage collection, partitions, object-oriented databases, storage reclamation.

Thesis Supervisor: Barbara H. Liskov
Title: NEC Professor of Software Science and Engineering

Acknowledgments

I would like to thank my thesis supervisor, Barbara Liskov, for her support and guidance. I learned a lot through my numerous discussions with her. Her careful reading and comments have greatly improved this thesis.

I am grateful to all the members of the Programming Methodology Group (Andrew, Atul, Bob, Dorothy, Eui-Suk, Joe, Liuba, Miguel, Paul, Ron, Sanjay, Quinton, and Umesh) for providing a friendly and interesting research environment.

I would like to express my gratitude for the following people:

Atul Adya, Joseph Bank, Miguel Castro, Jason Hunter and Joseph Bank for proof-reading my thesis.

Eui-Suk Chung for being a good friend and lunch companion.

Kei Tang for being a great friend and roommate. He is always there to listen to my problems and provide useful suggestions.

Special thanks to Rebecca for cheering me up during my difficult times. Her support and encouragement are greatly appreciated.

All members of my family for their unconditional love and support throughout my life.

This work was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Overview

Object-oriented databases (OODBs) provide persistent storage of objects with complex inter-relationships. They support transactions [10], a mechanism that allows client applications to group a set of reads and writes to objects as an atomic unit. The system ensures that either all or none of the operations within a transaction are executed in spite of system failures and concurrent access by other applications.

To prevent an OODB from running out of storage space, it is necessary to have a mechanism to reclaim storage of objects that are no longer used. Most OODBs rely on explicit deallocation of objects. Unfortunately, this is error-prone and places an extra burden on programmers. Failure to deallocate storage will result in permanent loss of space and deallocation errors may cause serious damage to the integrity of the database because of dangling references.

Garbage collection (GC) provides safe and automatic storage management of databases. While extensive research on garbage collection has been done in the context of programming languages[24], existing GC algorithms do not readily transfer for use in OODBs. Several issues have to be addressed for garbage collection to be practical in these systems:

Disk-resident data. The size of an object database can be very large (in gigabytes) and only a small portion of the database can be cached in main memory. The garbage collector has to minimize the number of disk I/Os and must also avoid replacing recently fetched objects in the cache.

Concurrency. The collection process must be able to run concurrently with client transactions. It should not affect the performance of client operations.

Fault tolerance. The collector has to preserve transaction semantics. It must survive system crashes and must not leave the database in an inconsistent state. In addition, the recovery process should remain fast even in the presence of GC.

We have designed and implemented a garbage collection scheme that addresses all the above issues. The work is done in the context of Thor [16], a distributed, client-server object-oriented database. Our scheme uses a partitioned approach [25]. The database is divided into *partitions* and each partition is collected independently. Our scheme uses a copying algorithm to collect each partition. Live objects are copied from a *from-space* to a *to-space*. The from-space is a single partition being collected and the to-space is a new, empty partition allocated by the garbage collector.

Our GC scheme has several advantages. (1) It is possible to make the partition size small enough so that the entire GC can be performed in main memory. This makes garbage collection fast and reduces the number of I/O operations executed by the collector. (2) The scheme is scalable because the amount of work done per GC is independent of the size of the database. (3) The collector is free to select which partition to collect. It has been shown that this can increase the amount of storage reclaimed [5]. (4) The collector runs concurrently with client activities. Minimal synchronization is required between the servers and clients. Client reads and updates always proceed with no extra latency whether the collector is running or not. (5) The scheme is fault-tolerant. Recovery is simple and fast even if the server crashes during GC.

To allow partitions to be collected independently, the database server has to maintain a list of inter-partition object references for each partition. We call this list an inter-partition reference list (IRL). The IRLs have to be maintained persistently because recomputing them after a server crashes involves scanning all the objects in the database. This slows down the recovery process significantly. This requirement raises a performance issue because every IRL update is a potential disk read and write. We present a new technique that reduces the number of I/Os required to update the IRLs. IRL update operations are deferred using the transaction log. The number of I/Os is reduced by processing the IRL updates in batches at some later time.

Currently, our scheme does not collect objects with inter-partition reference cycles. It is possible, though non-trivial, to augment the design using techniques such as migration of objects with cycles into the same partition or enlarging a partition to contain the objects with cycles. Another possibility is to use a complementary scheme based on global tracing [7, 11].

1.2 Related Work

Our work draws on previous research on garbage collection. Our goal is to design a garbage collector appropriate for large, persistent object storage systems. The main requirements imposed on garbage collection for these systems are very large storage space, concurrency, transactions, fault tolerance, and fast recovery. The garbage collector should also take into account the caching of database objects by clients, which is typical in a client-server environment.

We divide previous work on garbage collectors for persistent stores into two categories: non-partitioned GC and partitioned GC.

1.2.1 Non-partitioned GC

The research in this category focuses on designing a collector for persistent heaps that is incremental/concurrent, fault tolerant and preserves transaction semantics [6, 13, 21]. However, the approaches share one common problem. Each round of garbage collection requires the traversal of the entire persistent storage space. As a result, the schemes are not practical for very large databases. In addition, these works are done in the context of a centralized server and do not consider caching of objects at clients.

Two designs of atomic incremental garbage collectors for stable heaps are presented in Detlefs [6], and Kolodner and Weihl [13]. In both designs, the collector is closely integrated with the recovery system to make garbage collection recoverable. The recovery algorithm is complex in both cases and each step of collection has to be logged. In contrast, recovery in our scheme is simple and logging is only needed at the end of garbage collection.

O'Toole, Nettles and Gifford [21] describe a concurrent compacting collector for persistent heaps. They use a replicating garbage collection technique [20] to simplify recovery. In addition, transaction processing and garbage collection operations are decoupled using the transaction log. Our scheme uses a very similar technique to simplify recovery and increase concurrency. This work requires the garbage collector to concurrently process the entries in the log. Furthermore, a GC flip of the from-space and to-space can only occur when the redo log is empty. Our scheme does not have this requirement.

1.2.2 Partitioned GC

The idea of sub-dividing the address space into separate areas and collecting each area independently was first proposed by Bishop [4]. His work, however, was done in the context of a virtual memory system without persistence.

Generational GC [14] also divides the address space into separate areas, called *generations*, based on the lifetimes of objects. It differs from partitioned GC in several ways. First, generational GC exploits the fact that most objects have a very short lifetime and collects areas that contain newly created objects more frequently. Partitioned GC might use a different policy to select which partition to collect [5]. Second, generational GC only maintains inter-area references from young generations to old generations. As a result, collecting an old generation requires tracing all generations that are younger. Partitioned GC, on the other hand, can collect any partition independently.

Our scheme uses IRLs to keep track of inter-partition references. Similar data structures are used in the work by Bishop [4], in generational GC, as well as in distributed garbage collection algorithms to handle inter-node references [23, 18]. They are also used in the partitioned GC algorithms described below.

There are several recent works that address the issue of very large persistent storage spaces by using a partitioned approach for garbage collection [25, 5, 3]. However, none of these works addresses the I/O cost related to IRL modifications.

Yong, Naughton, and Yu [25] evaluate a number of GC algorithms for client-server persistent object stores. They propose an incremental, partitioned, copying GC algorithm. Their scheme is fault tolerant and recovery is simple due to the use of logical object identifiers. It requires the use of a write barrier at clients to detect modifications of object references. The new object references are recorded in a list, which is shipped back to the server at commit time. During garbage collection, locks have to be acquired on objects before they are copied to guarantee that the collector does not move an object that is being updated by an uncommitted transaction. Locking is not required in our scheme.

Cook, Wolf, and Zorn [5] investigate heuristics for selecting a partition to collect when a garbage collection is necessary. They show that one of its proposed partition selection policies is more cost-effective than the others. The policy is based on the observation that when a pointer is overwritten, the object it pointed to is more likely to become garbage. This work, however, does not describe the garbage collection algorithm itself.

Amsaleg, Franklin, and Gruber [3] describe an incremental mark-and-sweep garbage collector for client-server object database systems. The main focus of this work is to handle issues that arise due to the caching of disk pages by clients. The scheme ensures the correctness of a concurrent GC in spite of transaction rollback and partial flush of dirty pages from clients to the server. This problem does not occur in our system because only committed changes are sent back to servers. The paper briefly discusses how to extend the algorithm to collect partitions independently but details are not given.

Work that is closely related to this thesis is described in [18], which presents a distributed garbage collection scheme for Thor. That work concentrates on the coordination between multiple servers and clients to detect distributed garbage and turn it into local garbage, which can then be collected by local GC. However, it does not discuss the local GC. The garbage collector described in this thesis is designed to work in cooperation with the distributed GC algorithm in [18], providing a reclamation solution for distributed database systems.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 describes the system architecture of Thor. It introduces relevant concepts that will be used in later chapters. Chapter 3 presents an overview of our garbage collection scheme. It describes the data structures required to keep track of inter-partition references for independent GC of partitions. Chapter 4 discusses in detail the root entities required for GC of a partition. It provides an incremental root generation scheme. Chapter 5 identifies the cost associated with the maintenance of inter-partition references and describes an optimized scheme that reduces this cost. Finally, Chapter 6 concludes the thesis and suggests areas for future work.

Chapter 2

Thor Architecture

This chapter gives an overview of Thor, a distributed object-oriented database system. We will concentrate on the aspects of Thor that are relevant to this thesis. A more complete description of Thor can be found in [16, 15].

Thor is an object-oriented database management system that can be used in a heterogeneous, distributed environment. Thor provides a universe of persistent objects. It permits application programs written in various programming languages to share objects. Each object in Thor has a globally unique identifier called an *xref*. Objects contain data and references to other Thor objects. They are encapsulated so that an application using Thor can access their state only by calling their methods. Thor supports transactions that allow users to group operations so that the database state can remain consistent in spite of concurrency and failures.

Thor is partitioned into two distinct components: *Front Ends (FE)* running at client nodes and *Object Repositories (OR)* running at server nodes. An FE is created for each application program. It interacts with the application program and communicates with ORs to service client requests. An OR is responsible for managing the storage of persistent objects. There can be multiple ORs in the system (see Figure 2-1) and they can be geographically distributed. However, the physical location of objects is resolved transparently by Thor.

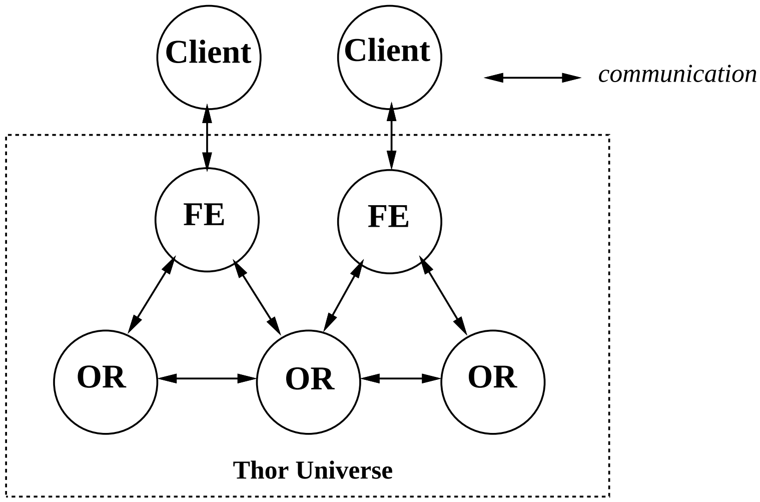

Figure 2-1: A configuration of clients, FEs, and ORs in Thor

2.1 Front Ends (FE)

When a client application using Thor initializes, it starts an FE process and establishes a connection with it. The FE acts as an external interface to the Thor universe. An FE communicates with the ORs to carry out the application's requests to access and modify objects. An FE *establishes a session* with an OR when it first connects with the OR. The FE and OR maintain information about each other until the FE closes the session (or until Thor shuts it down). We call an FE that has established a session with an OR a *connected FE* to that OR. In this thesis, we assume that an FE closes a session with an OR only when it shuts down or crashes. Thus, if an FE has ever fetched an object from an OR, it must maintain an open connection with that OR until it terminates. In addition, FE-client, FE-OR, and OR-OR communications are assumed to be reliable and messages are delivered in FIFO order. This can be implemented using well-known techniques such as timestamps, time-out, re-transmission and acknowledgements. Thor is designed to handle client/FE and network failures. If an OR is unable to contact an FE for a long time, it will properly shutdown the FE-OR session [18].

An FE usually runs on the same machine as the application. An application program never obtains direct pointers to objects. Instead, an FE issues *handles* that can be used to identify objects in subsequent calls. The FE maintains a *handle table* that maps from handles to objects. Handles are meaningful only during an application's particular session with its FE. An application program may be written in any programming language. The application and the FE usually run in separate address spaces to prevent corruption of persistent objects.

The FE is responsible for executing invocations of object operations by the client program. It *caches* copies of persistent objects and volatile objects created by the application. Objects in the FE cache may contain references to objects that are not in its cache. When an application makes a call in which an attempt is made to follow such a reference, a fetch request is sent to the OR containing that object. The OR sends the requested object to the FE along with some additional objects that might be required by the FE in the near future. This technique is referred to as *prefetching*.

An object fetched from a server contains references in the form of xrefs. The FE converts these references into virtual memory pointers to cached objects for better performance. This process is called *swizzling*. When an FE is about to commit a transaction, it converts the virtual memory pointers in modified cached objects back to xrefs before sending the modified versions of the objects to the ORs. This process is called *unswizzling*. To facilitate swizzling and unswizzling, an FE maintains a swizzle table, which is a two-way mapping from xrefs to virtual memory pointers.

Figure 2-2 illustrates how objects are accessed by a client using Thor. The FE caches copies of objects A and B from OR_1 and object D from OR_2. References to objects that are cached at the FE are swizzled into virtual memory pointers. The client is given handles to object A and B. These handles are stored in the client program's local variables *v1* and *v2*. The client uses handles to refer to Thor objects and to invoke operations on them.

2.2 Object Repositories (OR)

An OR manages a disjoint subset of the Thor object universe. Each OR has a globally-unique identifier, called the *OR id*. An OR contains a *root directory* object that contains references to other objects or directories in the Thor universe. The root directory is reachable from an internal, immutable root of the OR. We refer to that as the *OR root*. A client application can request the root directory object through the FE. After obtaining a handle to the root directory object, clients can navigate through its references to access other objects. For example, object A is a directory object in Figure 2-2.

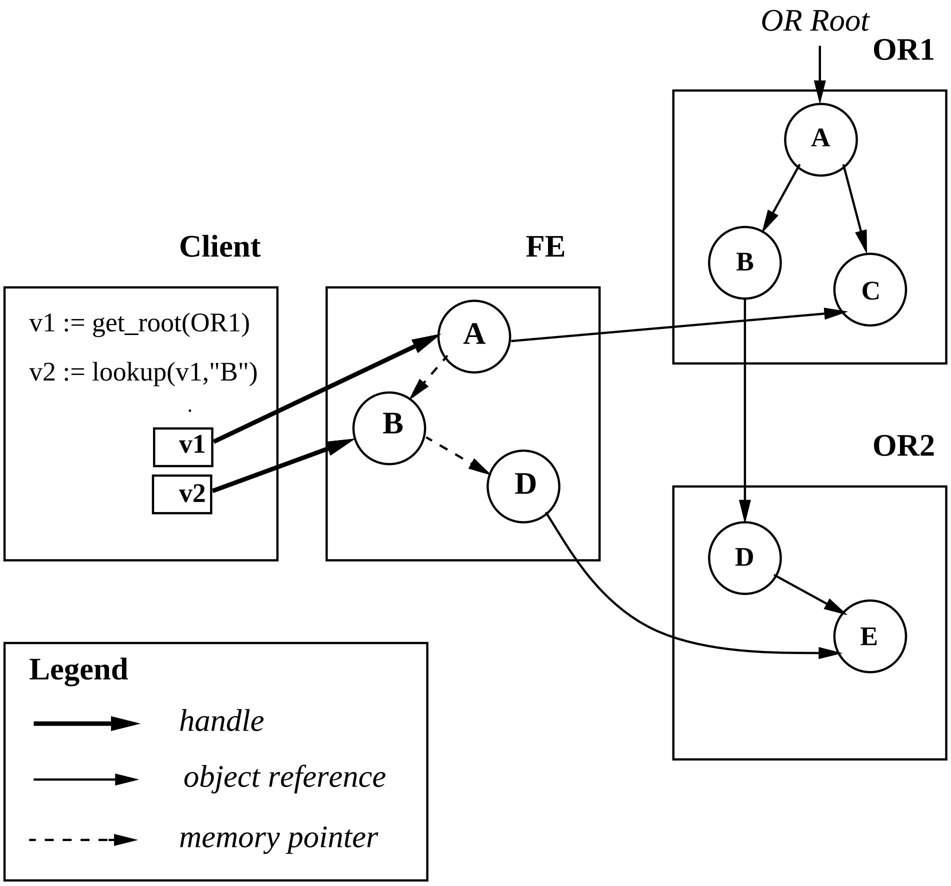

Figure 2-2: Object access by client and FE

2.2.1 Reachability and Garbage Collection

Persistence in Thor is based on reachability. An object is persistent if it is reachable from the root directory objects of the ORs or from other persistent objects. Client applications can create new volatile objects and these objects became persistent if they are made reachable from a persistent object or a root directory and the transaction commit succeeds. These objects are called *newly-persistent* objects. If a persistent object becomes unreachable from any persistent objects or from any handles in use in client applications, it becomes garbage and its storage will be reclaimed automatically.

Garbage collection in Thor is divided into two parts: local GC and distributed GC. The former works within individual ORs. It uses information maintained by the distributed GC and itself to reclaim storage. It may record extra information to assist the distributed GC. The latter is responsible for coordinating between different ORs as well as between ORs and

FEs to maintain a root set that protects all persistent objects and exposes the garbage to the local garbage collector. A design of a distributed GC protocol can be found in [18]. This thesis concentrates on the correctness and efficiency of the local GC protocol. However, we will also discuss some modifications to the distributed GC protocol described in [18].

2.2.2 OR Tables

For GC purposes, each OR maintains an OR table for every other OR, which records a conservative estimate of the incoming references from that OR. If OR_1 has a remote reference to an object in OR_2, the OR table for OR_1 at OR_2 has an entry for the referenced object. The use of OR tables is a form of reference listing [4, 18].

2.2.3 Segments

The storage in an OR is divided into a set of *segments*, which are similar to large disk pages. Each segment can store many objects. It is the unit of disk transfer and caching at an OR. Segments can be of different sizes. Each segment has a locally unique identifier called a *segment id*. Each segment has a header which contains an *indirection table*. The table is a mapping from an object index to the actual disk location within the segment.

Objects stored at an OR may contain references to objects at the same OR or at other ORs. A reference, also called an *xref*, consists of an OR id and and a local name within that server called an *oref*. An oref consists of a segment id and an object index. Given a reference, the corresponding object can be accessed by sending a request to the OR identified by the OR id. The OR then uses the segment id and the object index to locate the object efficiently. Migration of objects between different ORs is supported in Thor [1].

2.2.4 Partitions

The segments in an OR are logically grouped into sets of *partitions*. A partition consists of one or more segments. It is the unit of garbage collection. That is, each partition is collected independently of one another. The grouping of segments into partitions is based on some clustering strategy. The goal is to minimize the number of inter-partition references. Since a partition is just a logical grouping of segments, objects do not actually reside in partitions. Instead, they reside in segments. To avoid wordiness, we often use the term partition in this thesis when we actually mean the segment(s) of a partition.

2.2.5 Transaction Log

An OR uses a write-ahead log to implement atomic transactions by clients as well as low-level system transactions such as GC. In Thor, the transaction log resides in the main memory of an OR. Replication is used to allow the log to survive crashes [17]. The basic idea is that the log is replicated in the main memories of several machines. These machines are connected by a fast network and are backed up by uninterruptible power supplies. A log record is considered stable if it resides in volatile memory at all of these machines. Operations on an in-memory log are fast. For example, reading a log record is the same as a memory access and flushing a log record is equivalent to a network round-trip delay. In addition, the processing and discarding of log entries in an arbitrary order can be implemented efficiently.

2.2.6 Modified Object Buffer

The main memory of an OR is partitioned into a segment cache and a *modified object buffer* (MOB) [9], which contains the latest versions of modified or newly-persistent objects. In Thor, the modifications due to committed transactions are not immediately written to segments. Instead, they are appended to the transaction log and stored in the MOB. If the OR crashes, the modifications in the MOB can be reconstructed at recovery by scanning the log. Since the transaction log in Thor is implemented as a main-memory log, it is unnecessary for the log and the MOB to have two separate copies for the same object. Instead, they share the same objects in the main memory.

When an OR receives a fetch request from an FE, it first checks in the MOB if the latest version of the object is present due to recent modifications. If the object is not found in the MOB, the OR looks in the segment cache. If the object is not in the cache, the server reads the segment containing the object from disk, and stores the segment in the cache.

When the size of the log reaches a certain threshold, a separate background *flusher* thread will install objects from the MOB to the disk. We refer to this as *object installation*. To install an object, the flusher first fetches the segment where the object is located. If the segment is not in the cache, it has to be read from the disk. The object is then installed into the segment and the segment is written back to disk. The extra disk reads associated with object installation are referred to as *installation reads* [22]. Modified objects located in the same segment are installed together to reduce disk I/Os. Once the modifications have been written to the disk, they are removed from the MOB and the transaction log. The flusher thread runs independently of client activities. Thus object installation does not delay fetches or commits unless the MOB or the log fills up completely with pending modifications.

2.2.7 Transactions

To commit a transaction, an FE sends the transaction information to one of the ORs and waits for a decision. The OR that receives this commit request is called the *coordinator* of the transaction. All ORs affected by this transaction, including the coordinator, are referred to as *participants*.

Upon receiving a commit request, the coordinator executes a two-phase commit protocol [1, 12]. (The two phases can be combined to one if the transaction has used objects at only one OR.) In the first phase of the commit protocol, the coordinator sends a prepare message to each participant. A prepare message includes the new states of both new and modified objects belonging to a particular participant. Each participant decides whether it would commit the transaction and sends the decision (`commit` or `abort`) back to the coordinator. Before replying to the coordinator, each participant logs a *prepare* record in its stable transaction log. The prepare record contains the current states of the new and modified objects for that transaction at that participant.

If the coordinator receives a `commit` decision from all participants, it commits the transaction. Otherwise, it aborts the transaction. Then the coordinator informs the FE about the decision. In the second phase of the commit protocol, the coordinator notifies all participants about its decision. Each participant logs either a *commit* or *abort* record depending on the outcome of the transaction. If an abort record is present, the corresponding prepare record will be ignored. Otherwise, the modifications in the prepare record are entered in the MOB.

If an FE is notified that the current transaction has committed, it discards the *old* states of volatile objects that have been modified during the transaction from the cache. On the other hand, if the transaction has aborted, it discards the persistent objects that were modified by that transaction and restores the original states of the volatile objects. Volatile objects that were created during the transaction are treated differently. The creation of these objects is not undone. Instead they are restored to their initial states at creation.

This is because Thor does not invalidate the handles that have already been given to the clients. These handles may point to new volatile objects created during this transaction and the clients may continue to use them after the transaction has aborted.

Thor uses an optimistic concurrency control technique [2]. As a result, the objects cached at the FEs might be *stale* due to committed transactions of other clients. To reduce unnecessary aborts, an OR notifies FEs about stale objects. Each OR keeps track of information about the objects that have been cached at different FEs in a data structure called *FE table*. After an OR has installed a transaction's object modifications, it sends *invalidation messages* to FEs that may have cached these objects using the information in FE tables. Upon receiving an invalidation message for object x, an FE aborts the current transaction if its application has read or written x. It also discards x's copy in the FE cache. Any attempt to access the object x later will result in a fetch request to the OR containing the object x.

Chapter 3

Overview of Garbage Collection

This chapter briefly introduces the concepts and design of our garbage collection scheme. More details of the scheme will be discussed in later chapters.

Our scheme uses a partitioned approach: the segments at an OR are divided into disjoint partitions. Partitions are collected independently and only one partition is collected at a time. The garbage collector runs as a low-priority thread in the system. It maintains transaction semantics and survives server crashes. Client applications can continue to fetch objects from the server and commit transactions with no extra overhead during GC (see Section 3.4). Our collector uses a copying algorithm: it creates a new copy of the partition being collected and copies all its reachable objects into it. Objects are compacted within each segment of the new partition as they are copied. However, the collector never moves objects from one segment to another within the partition.

The collector only reclaims storage for objects that are unreachable at the *beginning* of GC. An object that becomes garbage during GC is not collected. It will be collected at the next GC cycle.

Object installation (Section 2.2.6) is *not* allowed during the entire GC. This can be done by suspending the flusher thread. However, the collector is allowed to install objects from the MOB to the segments of a partition before and after collecting that partition. Suspending the flusher thread during GC will not have a significant impact on the latency observed by clients because it does not lie in the critical path of client operations (unless the log or MOB is full).

The garbage collection of a partition p proceeds as follows: The collector first brings p up-to-date by copying objects for p from the MOB. Then it creates a new, empty partition and obtains the root set for GC of p. It traverses and copies all objects reachable from the root set to the new partition. After the traversal, the collector installs any objects for p that have been entered in the MOB during GC due to committed transactions. Finally, the collector atomically replaces the original partition with the new one. It also updates relevant inter-partition references information (see Section 3.1.1).

3.1 GC Roots

What are the roots for GC of a partition p? At the minimum, objects in p that are accessible from the OR root or client programs need to be protected against collection. Therefore, the root set includes the OR root and all references known at each FE. We will see in Chapter 4 that only a subset of these FE references needs to be included as roots. This subset of references is called the *volatile roots* of FEs.

The log in an OR contains new and modified objects of committed or prepared transactions. These objects have not been installed into the partitions yet and will not be traced directly by the collector. Therefore, all references in the prepare records of prepared and committed transactions in the log have to be part of the root set. Prepare records of aborted transactions can be ignored.

To protect objects referenced by remote ORs, the root set also includes the entries in OR tables, which record a conservative estimate of incoming references from other ORs.

In the root entities described above, the collector can safely ignore the roots that are not pointing directly into partition p. However, for each partition p, the OR has to maintain a list of incoming object references from other partitions to partition p. We call this list an *inter-partition reference list (IRL)*. The entries in the IRL of partition p are additional roots required for GC of p.

3.1.1 Inter-Partition Reference Lists

To keep track of inter-partition references, we use a coarse-grain reference counting scheme. The reference count of an object o is the number of other partitions containing reference(s) to o. A partition p contributes one to the reference count of an object o, where o is not in p, if it contains at least one reference to o. Otherwise, it contributes zero.

The IRL of each partition consists of an *inlist* and an *outlist*. The inlist of a partition p contains reference counts for all objects in p that are accessible from other partitions at the same OR. The outlist of p is a set of outgoing references from p to objects in other partitions in the OR. The states of the inlists and outlists only reflect the states of objects in the partitions. They do not take into account the new and modified versions of objects in the MOB. This does not violate safety because objects in the MOB are also contained in the prepare records in the log, which are part of the root set.

Each entry in an inlist consists of an oref and a positive reference count. Recall that an oref is an unique object identifier within an OR. The oref identifies an object that is referenced by other partitions and the reference count is the number of partitions that contains at least one reference to this object. When the reference count of an object becomes zero, it means no other partitions have references to the object and the object can be collected if there is no other reference to it. Therefore, the inlist of p provides *additional* roots needed to protect objects referenced by other partitions. Any oref in the inlist of p is an additional root for GC of p.

The outlist is a set of orefs of objects in other partitions. The presence of an oref in the outlist of a partition p indicates that p contains at least one reference to the object named by oref in another partition. The outlists are not used as GC roots. Instead, they are used to compute the correct reference counts for inlists of other partitions.

Figure 3-1 illustrates how the IRLs reflect the information about inter-partition references in partitions. Note that in partition q, even though the object C has two incoming references from another partition, the inlist entry for C still has a reference count of 1. This is because the two incoming references are from the same partition.

The inlists and outlists are maintained persistently because they are too expensive to recompute after crashes. They have to be updated whenever a partition is modified. This happens after object installation or garbage collection. In particular, new entries have to be added to the inlists and outlists after object installation to preserve safety because the installed objects may contain new inter-partition references. Section 4.1 and Chapter 5 provide more details on IRLs and how they are updated.

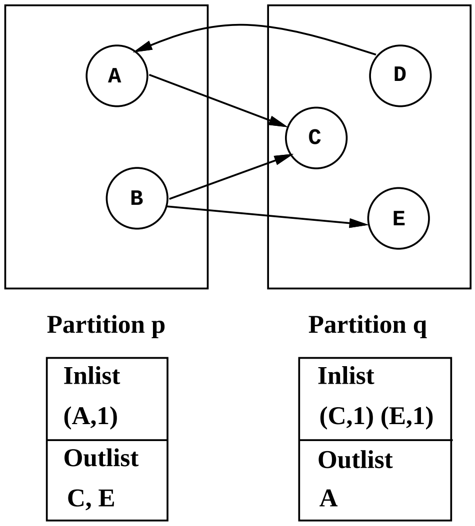

Figure 3-1: Inter-partition reference lists (IRLs)

3.2 Steps for Garbage Collection

The garbage collection cycle of a partition p consists of five steps:

Step 1: Install objects for p from MOB. When an OR is about to collect a partition p, it fetches the segments of partition p as well as the inlist and outlist of p. Then it installs objects for p from the MOB to make the partition up-to-date. Objects installed to p are not written to disk immediately and they are not removed from the MOB until the end of GC. Normally, when objects are installed from the MOB to a partition, new entries have to be added to IRLs if new inter-partition references are encountered in the objects installed. However, in this case the update of IRLs can be delayed because the collector will scan the whole partition for live objects during GC and all new inter-partition references will be discovered during the scanning process. Therefore, the update of IRLs can be done at the end of GC. This does not violate safety because no other partitions can be collected at the same time.

Step 2: Obtain the GC roots. To collect a partition p, the root set includes the volatile root sets in connected FEs and the *persistent roots*, which consists of the OR root, OR tables, objects in prepare records, and the inlist of p. Only the roots that are pointing into partition p are of interest. The root generation process is incremental and runs concurrently with client programs. During root generation, the collector does not allow entries to be removed from persistent roots. It sends a root-request message to each connected FE. It adds all roots returned by the FEs to the root set. Then it adds all entries in the persistent roots to the root set. This is described in more detail in Chapter 4.

Step 3: Traverse and copy live objects. After obtaining the root set, the OR creates a new, empty partition in volatile memory, which is referred to as the *to-space* for copying GC. The original partition being collected is called the *from-space*. The collector copies reachable objects from the from-space to the to-space. It also scans the objects and records any inter-partition references in them to construct a new outlist. The indirection table in the segments of the to-space partition allows the collector to find out easily whether an object has been copied. A mapping exists if and only if an object has been copied.

Step 4: Install objects for p from MOB. During GC, clients might commit transactions that cause new or modified objects for p to be entered into the MOB. These objects are not copied by the collector in Step 3. At the end of GC, it would be beneficial to incorporate these changes into the partition to avoid future installation cost. There are two differences between this step and Step 1. First, objects are installed into the to-space instead of the from-space. Second, the objects have to be scanned for inter-partition references and they are added to new outlist constructed in Step 3.

Step 5: Update Collected Partition and IRLs. During the GC tracing, the collector produces a new outlist that contains all the outgoing inter-partition references that the OR encountered. The collector uses the information in the original and the new outlist to determine which inlists of other partitions need to be updated. At the end of GC, the OR replaces the original outlist with the new outlist, updates the relevant inlists in other partitions, and replaces the original partition with the collected one. The replacement of the partition and update of IRLs have to be done *atomically*. See Section 4.1.2 for more detail.

3.3 Basic Correctness

Because of concurrent activities in the system, an important correctness condition for this scheme is that no new roots for the partition being collected will come into existence after the root generation (Step 2). Note that the flusher thread is the *only* mutator of segments (and thus partitions) besides the collector itself. Client commits do not directly mutate segments because committed transactions only change the log and the MOB, but not the contents of the segments. Since the flusher thread is suspended during GC, the partition being collected is never mutated except by the collector itself.

As a result, once a *correct* snapshot of roots is generated, that snapshot is guaranteed to protect all reachable objects for the entire duration of GC even in the presence of client fetches and commits. Any live object that is reachable from the snapshot of roots will continue to be reachable during the entire GC and any object that is not reachable from the root set cannot be turned into being reachable again.

3.4 Concurrency and Recovery

This scheme allows client operations to proceed concurrently with no extra overhead during GC. Since the garbage collector only modifies the to-space partition, it is decoupled from the client fetches and commits. FEs can continue to fetch objects from the segment cache, the MOB, and the from-space in the case of objects in the partition being collected. Transaction commits can proceed because modified objects are just appended to the log and entered in the MOB. Synchronization is only required at the end of GC when the segments of the from-space partition are atomically replaced by the segments of the to-space partition (Step 5).

Using a copying GC algorithm makes the GC recovery simple. If the server crashes during GC before Step 5, the collector can simply discard the to-space and start over. No recovery is required on the from-space because it has not been modified by the collector. See Section 5.6.2 for more detail on recovery.

Chapter 4

GC Roots

As explained in Section 3.1, the roots for GC of a partition p include the OR root, references in prepare records, entries in OR tables, the inlist of p, and the volatile roots of connected FEs. In this chapter, we will describe some of these root entities in detail. We will also discuss when they have to be updated.

4.1 Inter-Partition Reference Lists

The safety of the partitioned garbage collection scheme depends on the timely updates of the IRLs. There are two invariants on the IRLs that need to be maintained by the local GC protocol:

- For any valid partition p, the inlist of p contains the reference counts of all objects residing in p. In particular, the reference count of an object o in p is equal to the number of partitions other than p at the same OR that contain at least one reference to object o.
- For any valid partition p, the outlist of p is a set of references. If there is an object in p that contains an inter-partition reference x, then the reference x exists in the outlist of p.

To maintain the above invariants, the flusher thread has to be *extended* to update the IRLs when it installs objects from the MOB to segments. In addition, the IRLs have to be updated after GC of a partition. However, the IRLs do *not* have to be modified when clients commit transactions because new and modified objects are not written to partitions directly. They are entered into the MOB and the relevant IRLs will be updated when the objects are installed into the partition later. As explained in Section 3.1.1, this does not violate the GC safety.

The IRL update scheme described below illustrates how the IRLs need to be updated by object installation and garbage collection. An optimized version of the scheme is discussed in Section 5.4.

4.1.1 Object Installation

New entries have to be added to the IRLs when modifications to objects in p are installed because this might result in new inter-partition references. Specifically, before the flusher thread installs an object from the MOB to a segment belonging to a partition p, it needs to scan the object for inter-partition references. For each inter-partition reference oref, the flusher has to determine if p already contains this reference by checking whether the outlist of p contains oref. If oref is not present in the outlist of p, the flusher needs to increment the reference count of oref in the inlist of partition q, where q contains the object being referenced. It also needs to add oref to the outlist of p. On the other hand, if oref is present in the outlist of p, nothing needs to be done because the reference count for oref already reflects the reference from p.

Figure 4-1 illustrates the object installation process. There are two modified objects A and F in the MOB due to a committed transaction. When the flusher installs object F to partition q, it checks the outlist of q and determines that the reference to C from object F is a new inter-partition reference. As a result, it has to add C to the outlist of q and increment the reference count of C in the inlist of p. On the other hand, when the flusher installs object A to partition p, it does not need to update the IRLs because reference D is already in the outlist of p. A and F are removed from the MOB and the log after they are installed.

4.1.2 Garbage Collection

In traditional reference counting schemes, when a reference is deleted, the corresponding reference count is decremented. However, in our case, an inter-partition reference being deleted from an object does not necessarily mean that the OR can decrement the corresponding reference count. This is because the reference counts maintained by the OR record

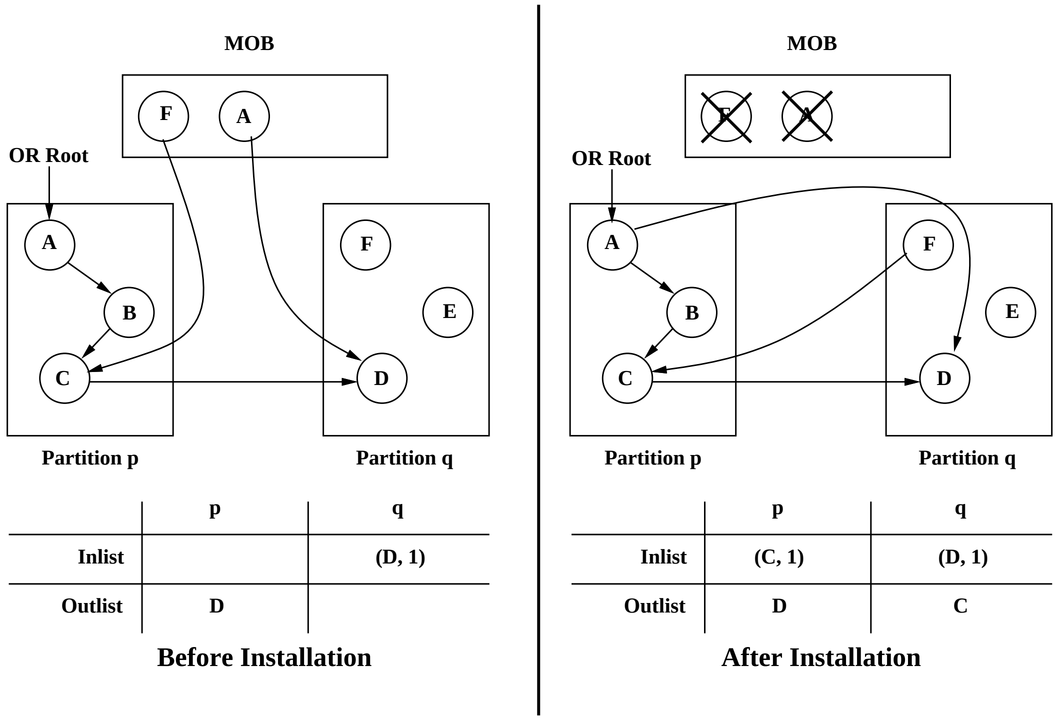

Figure 4-1: IRL update due to object installation

the number of referencing partitions, not the number of referencing objects. To determine whether the last inter-partition reference is being deleted from a partition p, the OR needs to scan all the objects in p. After scanning, the OR can construct a list of outgoing references to other partitions, that is, the updated outlist. By comparing the new outlist with the original one, the OR can decrement corresponding reference counts for inter-partition references that no longer exist.

Obviously, it is not practical to scan the whole partition every time a reference is deleted from an object. Instead, the OR can perform the scanning lazily because the safety of the GC protocol is not violated by not decrementing the reference counts immediately. A good time to scan a partition p is when the OR performs GC on p.

At the end of GC, the OR compares the new outlist with the original outlist of p. If an oref is present in the original outlist and not in the new outlist, this means that p used to contain this reference but no longer does. Therefore, the OR can safely decrement the inlist entry for the oref. On the other hand, if an oref is present in the new outlist but not in the original outlist, this means that oref is a new inter-partition reference in p. This can happen because the IRLs are not updated immediately when the collector installs objects from the MOB at the beginning of GC (Step 1 in Section 3.2). In this case, the OR needs to *increment* the inlist entry for that oref. Finally, the original outlist in disk is replaced by the new one and all modified inlists are written to disk.

Figure 4-2 illustrates how the IRLs are updated after GC of partition q. Objects D and E are garbage collected and the new outlist constructed by the garbage collector only contains B. By comparing the new outlist with the original one, the garbage collector determines that it must decrement the inlist entry for C in partition p. Then it replaces the original outlist of q with the new one.

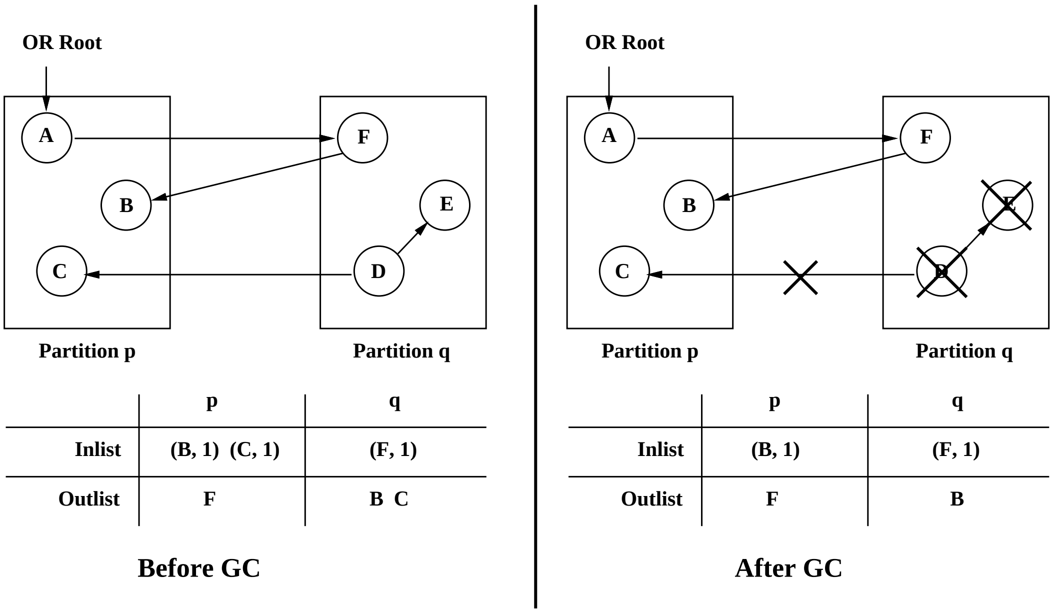

Figure 4-2: IRL update due to garbage collection

4.2 OR Tables

Each OR OR_1 maintains an OR table for every other OR OR_2, which records a conservative estimate of incoming references from OR_2 to OR_1. An OR table is a set of xrefs. The invariants on OR tables maintained by the GC protocol are:

- If OR_2 has a remote reference to an object in OR_1, then an entry for the referenced object exists in the OR table for OR_2 at OR_1.

- If an FE has a reference to an object in OR_1 and it is not connected to OR_1, then an entry for the referenced object exists in the OR table at OR_1 for some other OR connected to the FE.

To maintain the first invariant, entries have to be added to the OR table at OR_1 for OR_2 when OR_2 receives a new remote reference to OR_1 due to committed transactions. The insertion of entries has to be incorporated in the first phase of the two-phase commit protocol. More details can be found in [18].

To maintain the second invariant, before an OR OR_1 can inform another OR OR_2 about the deletion of a remote reference xref, it has to first ensure that the xref does not exist in any FEs it has a connection with. This can be accomplished by sending a message to each connected FE to inquire about the existence of the xref in the FEs. It only informs OR_2 if the xref is not present in volatile roots of all connected FEs.

To shrink the OR tables, it is necessary for an OR to discover it no longer has a remote reference to another OR and inform the other OR about it. This can be done by keeping a *remote outlist* for each OR. Each entry in the remote outlist consists of an *xref* and a list of *partition ids*. The xref identifies an object in a remote OR that is referenced and the list of partition ids identifies the partitions at this OR that contain references to that object.

During GC of a partition p, the collector constructs a new list of remote references it encounters during tracing. At the end of GC, it compares this list with the remote outlist. If the remote outlist indicates that p contains an xref x but x is not present in the list of remote references, then p is removed from the partition list associated with x in the remote outlist.

After updating the remote outlist, the OR scans the remote outlist for xrefs that have an empty partition list. As mentioned above, the OR first asks all connected FEs about the existence of these xrefs in the FEs. If an xref is not present at any FE, the OR informs the other OR about the deletion of this reference. When the OR receives an acknowledgement of the deletion, it removes the xref from the remote outlist. Otherwise, the xref will be kept in the remote outlist until the OR determines that no connected FE contains that reference.

4.3 FE Volatile Roots

A reference must be in the volatile root set of an FE if it can become the only reference to a persistent object. In a very conservative approach, the volatile root set for a partition p would include every reference to p that is known to any FE in the Thor universe. Obviously, this is not practical. First, this can result in a very large volatile root set since an FE might cache a lot of objects due to prefetching. Second, the OR may not even know the existence of an FE that contains a reference to it because that FE may have obtained the reference from an object in another OR. Thus, it has never established a connection with this OR.

It will be shown that the volatile root set for a partition p only needs to include information about references in FEs connected to the OR, and for these FEs, it only needs to include entries in the handle table, references in reachable volatile objects, and references in new versions of persistent objects modified by the current transaction. Both old and new versions of volatile objects have to be scanned because the current transaction may abort or commit. We need to include references in new versions of modified persistent objects because the references in these new versions may no longer be found in the handle table or volatile objects and the OR does not learn about these modified objects until the transaction commits. In this definition of volatile roots, it is unnecessary to include the references in an *unmodified* object cached at any FE. It is also unnecessary to include volatile roots in FEs that do not have a connection to the OR doing GC (see Section 4.3.1 and 4.3.2).

When an OR is about to perform GC, it first sends out all pending invalidation messages (Section 2.2.7). Then, it sends a root-request message to each connected FE. Upon receiving the root-request message, an FE first processes all pending invalidation messages from ORs. Then it performs a local collection of its volatile object heap. This collection should be fast because it only involves the volatile heap. It is also possible for the FE to avoid the GC by keeping root information approximately up to date: the information is recomputed during FE GC, when transactions commit, when handles are given to client programs and when new volatile objects are created.

During FE GC, all reachable volatile objects are scanned and the references to persistent objects in them are returned as roots. In addition, the entries in the handle table and references in modified persistent objects are returned.

4.3.1 Omitting Unmodified Persistent Objects

We argue that omitting references in unmodified objects cached at FEs does not violate GC safety. Suppose an FE has cached an unmodified object x. Object x contains a reference to an object y in the partition of OR_1 being collected. Object x may be located at OR_1 or at some other OR OR_2. In addition, the object x at the OR may either be identical to the version cached at the FE or it is a newer version. It cannot be an older version because the FE has not modified x. In either case, the FE is connected to the OR containing x.

Object x has a newer version at OR:

- Figure 4-3 illustrates the situation where x is located at OR_1. The FE is connected to OR_1 and has an unmodified copy of x in its cache and x contains a reference to y. x is updated at OR_1 (because of the commit of a transaction running at a different FE) and the reference to y is removed. Before the FE receives a root-request message from OR_1, it will receive an invalidation message for x. Therefore, it will discard x from the cache and the client will not have access to x and thus the reference to y.

- If x is located at some other OR OR_2 (Figure 4-4), y is initially protected by the OR table at OR_1. In this case, FE is connected to OR_2. If x is updated at OR_2 resulting in the deletion of the reference to y, the key question is whether y is still protected by the OR table. A problem arises only if the entry for y is not in the OR table anymore. But this means OR_2 must have performed a local GC, and before it did that GC it would have delivered an invalidation message to the FE, causing the FE to discard x from its cache.

Object x is identical at FE and OR:

- If x is located at some other OR OR_2, then y is protected by the OR table for OR_2.

- If x resides in the MOB of OR_1 and has not been installed yet, then y is protected by the prepare records in the log.

- If x is located at OR_1 but in a different partition, then y is protected by the inlist of the partition being collected.

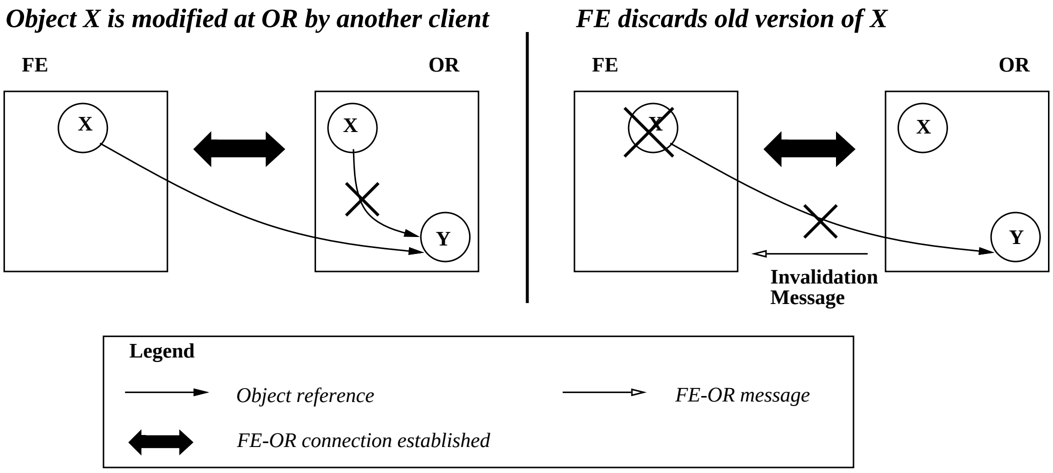

Figure 4-3: Invalidation of an object involving one OR

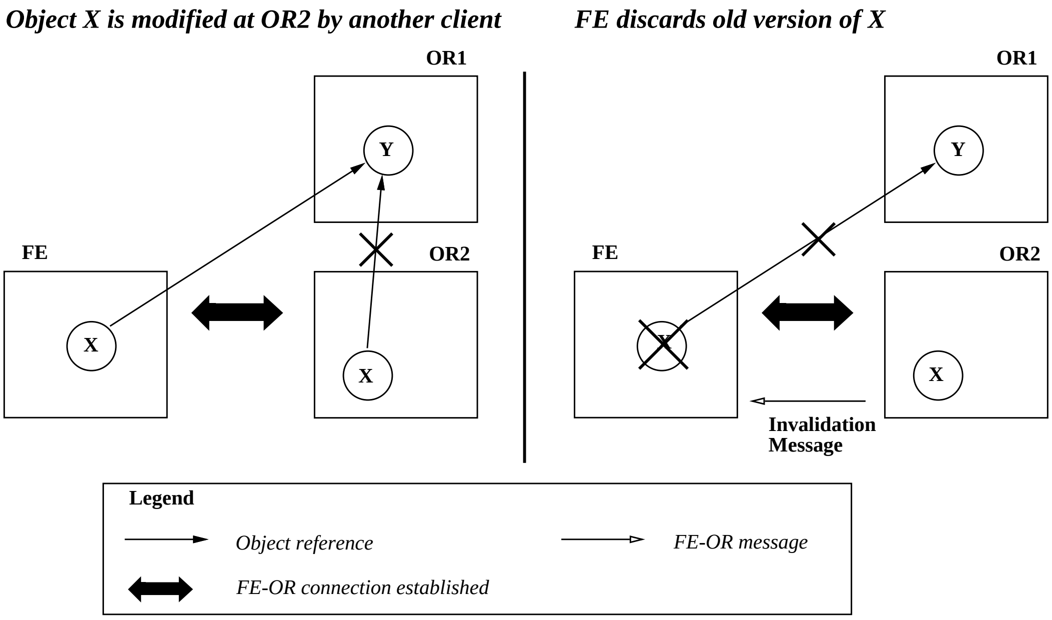

Figure 4-4: Invalidation of an object involving two ORs

- If x is located in the partition being collected, we need to analyze whether x will be traced during GC. If it will be traced, then y is protected trivially. Otherwise, the only place where a reference to x may exist will be in another unmodified object cached in the FE. And we can use the above argument recursively to show that x is not accessible to the client.

4.3.2 Omitting Unconnected FEs

Consider the FEs that are not connected to an OR OR_1 doing GC. Suppose one of these FEs has a reference r when the root snapshot is taken and r points to an object in OR_1. If r is contained in an unmodified object, it can be omitted (Section 4.3.1). Otherwise, r is in the volatile root set of the FE.

The FE must have obtained r from an object from another OR OR_2 and the FE must be connected to OR_2. The OR table at OR_1 for OR_2 must have had an entry r at some point in the past. If r is not in the OR table when the snapshot is taken, then a delete message must have been sent from OR_2 to OR_1 to remove r from the OR table. But before sending such a delete message, OR_2 must have asked all its connected FEs whether r is present in their volatile root set. Since r is present at the volatile root set, OR_2 will not send the delete message. This is a contradiction. Therefore, the OR table at OR_1 must have an entry for r when the snapshot is taken and the object referenced by r will be protected from collection.

Figure 4-5 illustrates the above argument. OR_2 has performed GC and discovered that it does not hold a reference to object y anymore. However, before sending a delete message to OR_1, it first sends a query message to all connected FEs. One of the FEs informs OR_2 that it still contains a reference to y. As a result, the delete message will not be sent.

4.4 Incremental Root Generation

It is expensive to stop all ORs and FEs and generate the roots atomically. In fact, it is not necessary to do this. We present an incremental root generation scheme that allows concurrent object fetches and transaction commits during root generation. There are three steps for the root generation scheme:

Step 1: From this point on until the end of root generation, no entries are allowed to be deleted from the persistent roots. The persistent root set includes the OR tables, the

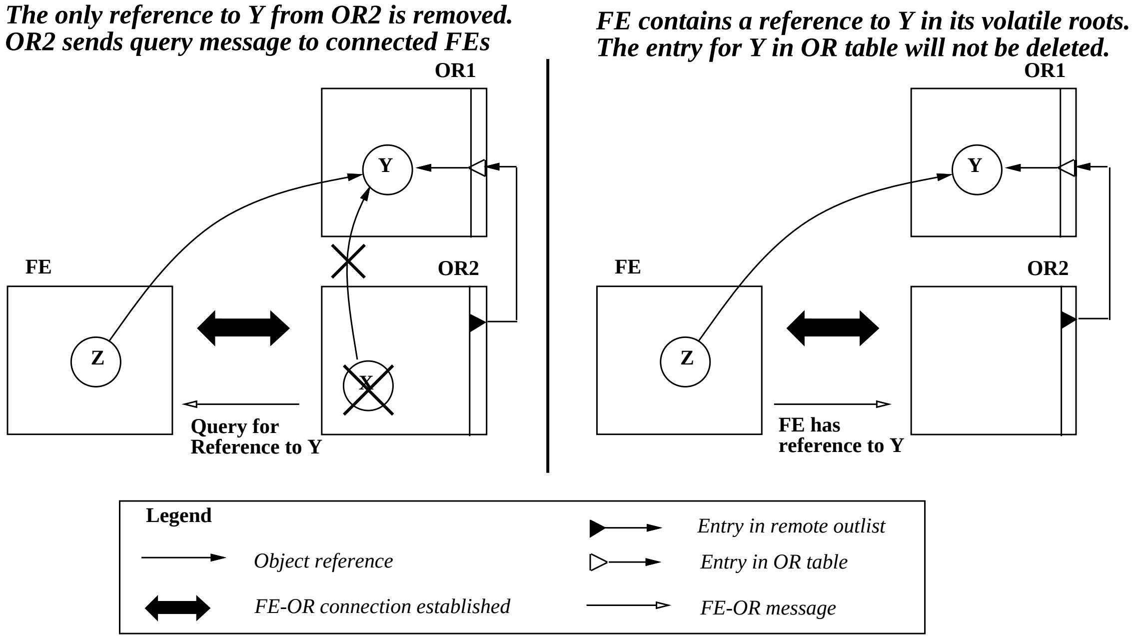

Figure 4-5: Protection of references in unconnected FEs by OR tables

log, and the inlist of p. New entries are still allowed to be *added* to the persistent roots.

Step 2: The OR sends a root-request message to each connected FE. Each FE returns its volatile root set, which is added to the root set.

Step 3: After the OR has received responses from all connected FEs, the OR obtains the persistent roots in the following order: root directory object, inlist of p, OR tables, entries in the log.

4.4.1 Correctness

It is crucial for correctness to request the volatile roots from FEs before scanning the persistent roots. To see why this is the case, suppose the collector instead scans the persistent roots before requesting volatile roots from FE. Then the following erroneous scenario might occur: An FE caches and modifies an object x to hold the only reference to object y. The collector scans all the persistent roots and does not find any reference to y. Then the client commits the transaction, which makes the modification to x persistent and terminates after the commit. The collector now sends out root-request messages to FEs. However, it will never receive a reference to y as a root because the client has terminated already. Thus, object y will be collected incorrectly.

This scheme also depends on the fact that the clients in Thor cannot communicate and ship modified objects or object references directly. Object updates must be stored at ORs before another client can observe them.

The key correctness condition for the root generation scheme is that if an FE (connected or unconnected) obtains a *new* root pointing into partition p after Step 1, then either the root points to a newly-persistent object or the root must be part of the persistent roots before Step 3 begins.

If the new volatile root points to a newly-persistent object, the newly-persistent object will be appended to the log and entered in the MOB when the transaction commits. The newly-persistent objects will survive the collection because objects in the MOB are not garbage collected.

Otherwise, the FE must have copied the reference r from an *unmodified* object in its cache. If that object is already in the cache when GC begins, Section 4.3.1 shows that it is okay to ignore this reference. Otherwise, the reference r must exist somewhere in the Thor system when GC begins. Either the reference is in the volatile roots of an FE or there is a live object containing the reference r at some OR.

Consider the former case:

- If the FE is connected to the OR, the collector would obtain the reference r as a root when it requests volatile roots from the FE in Step 2.
- Otherwise, the reference r is protected by the OR tables because of the second invariant for OR tables (Section 4.2). No entries are removed from the OR tables after GC begins. So the collector would obtain r when it scans the OR tables.

For the latter case:

- If the object belongs to a different OR, then r is in the OR table. No entries are removed from the OR tables after GC begins. So the collector would obtain r when it scans the OR tables.
- If the object resides in the MOB of the same OR and has not been installed yet, reference r is protected by the prepare records in the log, which will be scanned by

the the collector in Step 3.

- If the object belongs to a different partition in the same OR, then r is in the inlist of p. No entries are removed from the inlist after GC begins. So the collector would obtain r when it scans the inlist.

- If the object belongs to the same partition, the reference r is protected by tracing. This is because when GC begins, the object is live (or the FE won't be able to access it) and will remain so because modifications are not installed while the partition being collected.

For the prepare records in the log, it is not necessary to scan the entire log for roots. Instead, the collector only needs to scan all the records from the beginning of the log up to the point when the OR receives all replies for the root-request messages. Any new prepare records appended after that point only contain roots that either points to newly-persistent objects (which are not collected) or to objects that are already protected by the root entities scanned.

Chapter 5

Inter-Partition Reference Lists

5.1 I/O Overhead for IRL Updates

The IRLs have to be maintained persistently because recomputing them after server crashes would require scanning *all* the objects in the database. This makes the recovery process too expensive and slow.

Special system objects are used to store the inlists and outlists in the disk. Depending on the number of inter-partition references in an OR, the total size of the IRLs can be quite large. Hence, it is not practical to cache all IRLs in the main memory.

In a naive implementation, every time the OR needs to increment or decrement a reference count for an oref, it has to fetch the corresponding inlist, make the change, and write it back to disk. During the object installation of a partition p, the OR has to fetch the outlist of p to determine whether a new inter-partition reference is encountered. It may also need to update the outlist as well as inlists of other partitions. All of these updates can potentially generate a large number of disk reads and writes, which raises a performance issue. In this chapter we present an optimization that reduces the number of I/Os required to modify the IRLs.

5.2 Deferred IRL Operations

The basic idea behind this optimized scheme is to delay the updates to the IRLs by queueing the IRL operations to be performed. The queued IRL operations are stored in the stable log and are processed in a batch later. By delaying these operations, the OR can perform re-ordering of operations to reduce the number of I/Os provided the re-ordering does not violate the correctness or consistency of IRLs.

There are three kinds of IRL operations that need to be queued. Their semantics is summarized in Figure 5.2. For each operation, the "requires" clause specifies the pre-condition of the operation. The "modifies" clause specifies which IRL is updated, and the "effects" clause specifies what will be updated when the operation is processed at some later time.

The `IncrCount` and `DecrCount` operations are used by the garbage collector to change the reference count of an inlist entry. The `InterRef` operations are used when inter-partition references are encountered during object installation. The `IncrCount` operations are also used when the `InterRef` operations are processed. More detail can be found in Section 5.4.

```
IncrCount(Partition p, Oref x)
// Requires: object x resides in partition p
// Modifies: inlist of p
// Effects:  Increment the reference count of x by 1

DecrCount(Partition p, Oref x)
// Requires: object x resides in partition p
// Modifies: inlist of p
// Effects:  Decrement the reference count of x by 1

InterRef(Partition p, Oref x)
// Requires: object x does not reside in partition p
// Modifies: outlist of p
// Effects:  If outlist of p contains x, do nothing. Otherwise, add x
//           to the outlist and replace this operation with an
//           IncrCount(q, x) operation, where q is the partition that
//           contains the object named by x
```

Figure 5-1: IRL update operations

One potential problem with delaying the IRL operations is that the states of the IRLs are not up-to-date unless there are no pending IRL operations queued. In the OR, up-to-date states of IRLs are only needed when the OR is about to perform GC. In Section 5.5, we will describe a scheme to obtain a conservative estimate of IRLs that is accurate enough for GC purposes. As a result, it is not crucial to always maintain IRL states up-to-date and this allows the OR to delay processing IRL operations.

5.3 Re-ordering of IRL Operations

We argue that given a sequence of `IncrCount`, `DecrCount`, `InterRef` operations, the *final* states of the IRLs after processing all the operations in the sequence are *independent* of the order in which the OR processes those operations.

Consider the `IncrCount` and `DecrCount` operations. The outcomes of these operations do not depend on any `InterRef` operations. These operations on different inlist entries do not affect one another. For operations on the same inlist entry, they can be freely re-ordered provided a negative reference count cannot occur. In our implementation, the OR is going to process `IncrCount` and `DecrCount` operations of the same object in the same order as they are queued. As a result, a negative reference count will never appear in the inlists.

Now consider the `InterRef` operations. An `InterRef(p, x)` operation has two possible outcomes, depending on whether x is present in the current state of the outlist of p. We need to ensure that the same outcome is generated in the re-ordered sequence of operations. Note that `InterRef` is the *only* operation that can modify the outlists. As a result, the outcome of an `InterRef` operation will not be affected by the ordering of `IncrCount` and `DecrCount` operations. Furthermore, `InterRef(p, x)` is the only operation that can modify the presence of x in the outlist of p. So the outcome of an `InterRef(p, x)` operation will not be affected by the ordering of other `InterRef(q, z)` operations, where q $\neq$ p or $z \neq x$. Finally, the order of evaluation for a sequence of identical `InterRef(p, x)` operations does not matter. In fact, a sequence of identical `InterRef(p, x)` operations is equivalent to one `InterRef(p, x)` operation.

5.4 I/O Optimized Scheme for IRL Updates

We now present an I/O optimized scheme for updating the IRLs. The queued IRL operations are compacted by removal of redundant entries and re-ordered to reduce the number of I/Os. The flusher thread and the garbage collector need to be augmented. In addition, a new thread, called the *IRL update thread*, needs to be added to process the IRL operations queued in the log. This thread is responsible for applying the queued IRL operations to the actual inlists and outlists. The execution of object installation, garbage collection and IRL updates do not interleave and they are never executed concurrently. Issues related to recovery are discussed in Section 5.6.

5.4.1 Object Installation

Before the object installation, the flusher first initializes an empty buffer in server memory to record IRL operations. As the flusher installs modified objects from the MOB to segments, it scans for inter-partition references contained in the objects. For each inter-partition reference x in the object, the flusher records an `InterRef(p, x)` operation to the buffer. After all the objects in the MOB are installed, the flusher writes a single log record containing all the `InterRef` operations recorded in the buffer and the ids of the installed segments. Then it removes the installed objects from the MOB and the log. As a result, the object installation process only requires one stable write to the log.

5.4.2 Garbage Collection

Before performing GC on a partition p, the collector reads the inlist of p and applies all the queued `IncrCount` and `DecrCount` operations for that inlist to obtain the new inlist. It also processes `InterRef` entries to obtain additional roots (see Section 5.5).

After GC of a partition p, the collector has constructed a new outlist, which contains all the outgoing references from p to other partitions. It first initializes an empty buffer in server memory to record IRL operations. Then it compares the new outlist with the original outlist of p to determine what inter-partition references are removed by GC and what inter-partition references are in p but were not reflected in the old outlist. For each inter-partition reference x that is removed, the collector appends a `DecrCount(q, x)` operation to the buffer, where q is the partition containing x. For each inter-partition reference x that is in the new outlist but not in the original one, the collector appends a `IncrCount(q, x)` operation to the buffer, where q is the partition containing x.

After recording all `IncrCount` and `DecrCount` operations, the collector is ready to write out the IRL operations, the new inlist and outlist of p, and the collected partition. For recovery purposes, it is important to atomically replace the original segments of the partition with the collected ones. Two techniques can be used: If the ORs are replicated to provide high availability, different replicas can cooperate such that at least one replica contains the contents of the original partition, which can be restored if a crash occurs while the partition is being written. Alternatively, the collector can use non-volatile memory (NVRAM) to achieve atomicity. The size of NVRAM has to be large enough to hold the segments of one partition.

After writing out the partition the collector appends a log record containing the IRL operations, the new inlist and outlist, and the id of the partition being collected. The presence of this end-of-GC record indicates the completion of GC. This record supersedes the queued `InterRef` operations for inter-partition references originating from p, `IncrCount` and `DecrCount` operations for the inlist of p, and modified objects installed for p, and they are removed from the log. The `InterRef` operations for p are superseded because the outlist computed after GC accurately reflects the inter-partition references contained in the objects of the partition and all the necessary `IncrCount` operations have been queued.

5.4.3 IRL Update Thread

When the number of queued IRL operations in the log reaches a certain threshold, the IRL update thread wakes up and updates the IRLs by processing the operations queued in the log. Since the IRLs are updated together in batches, the IRLs are clustered together on disk rather than stored with the partitions to reduce I/Os.

The IRL update thread first updates the outlists by processing the queued `InterRef` operations. Then it updates the inlists by processing the queued `IncrCount` and `DecrCount` operations. Note that it is not necessary to process all the queued IRL operations in the log. Instead, the update thread can choose to process a portion of the log.

To update the outlists efficiently, the IRL update thread scans the log for `InterRef` operations and sorts the operations by partitions (the first argument of `InterRef` operations). For each outlist of partition p that has at least one `InterRef` operation, the thread fetches the outlist, and applies the `InterRef` operations related to this outlist to obtain the new outlist. Applying an `InterRef(p, x)` operation involves checking whether x is present in the outlist of p. If so, do nothing. Otherwise, add x to the outlist and append an `IncrCount(q, x)` operation to a volatile buffer, where q is the partition that contains object x. After applying all `InterRef` operations, the IRL update thread writes a single record to the log. That record contains the `IncrCount` operations recorded in the buffer, all the outlists that are updated, and the indices of the log records that have been processed.

The presence of this record causes all processed `InterRef` operations to be removed from the log.

To update the inlists, the update thread scans the log for `IncrCount` and `DecrCount` operations and sorts the operations by partitions (the first argument of the operations). For each inlist of partition p that has at least one `IncrCount` or `DecrCount` operation, the thread fetches the inlist, and applies relevant `IncrCount` and `DecrCount` operations to obtain the new inlist. After applying all inlist operations, the IRL update thread writes a single log record containing all the updated inlists and the indices of the log records that have been processed. The presence of this record causes all processed `IncrCount` and `DecrCount` operations to be removed from the log.

5.5 Conservative Inlist Estimate

The entries in the inlist of partition p with positive reference counts are treated as roots when the OR wants to GC partition p. In the I/O optimized scheme, however, the inlists may not contain the most accurate reference counts unless there are no queued IRL operations. Obviously, it is undesirable to apply all queued IRL operations to the IRLs on disk whenever the OR wants to perform GC.

To garbage collect a partition p, the OR only needs to determine which entries in the up-to-date inlist of p have a positive reference count. It does not need the actual value of the reference counts. Instead of evaluating all queued IRL operations, the OR can compute a conservative estimate of the inlist by looking at the queued IRL operations that might affect the state of inlist of p. These include `IncrCount` and `DecrCount` operations with p as the first argument and `InterRef(q, x)` where x is a reference pointing into p.

To determine which entries in the up-to-date inlist would have positive reference counts, the OR first fetches the inlist of partition p. Then, the OR applies all the queued `IncrCount` and `DecrCount` operations of p in the log. After applying the operations, any entry in the inlist with reference count greater than zero is added to the root set. The new inlist will be written to the log at the end of GC.

In addition, for each `InterRef(q, x)` operation in the log, where q is not equal to p and x is an object contained in partition p, the OR adds x to the root set. This is because a queued `InterRef(q, x)` indicates that partition q contains a reference to object x. As a result, x has to be protected.

5.6 Recovery Issues

It is important that deferring the IRL updates does not result in incorrect IRL states after server crashes. The deferred IRL update operations are stored stably in the log and will survive server crashes. However, it is important that the OR does not incorrectly write or apply these IRL operations more than once because they are not idempotent. To simplify the discussion, we assume a single record can be entered into the log *atomically* regardless of the record size. This can be implemented easily by using a record to mark the end of a series of log records that have to be appended as an atomic unit.

5.6.1 Object Installation

As explained earlier, the flusher records the IRL operations in a volatile buffer as it moves modified objects from the MOB to segments on disk. Objects are not removed from the MOB or the log at this point. If the server crashes, the IRL operations in the buffer will be discarded and the flusher can start over after the MOB is reconstructed from the log.

After the flusher has completed installing modified objects in the MOB, it appends a single record to the log. This record includes all the IRL operations recorded in the buffer and the ids of the installed segments. The presence of this log record indicates that all the modified objects in the segments recorded in this log record have been installed. If the server crashes at this point, the flusher will not not attempt to install the modified objects for the installed segments again.

5.6.2 Garbage Collection

Recall that the collector first writes the collected partition to disk (atomically) and then writes an end-of-GC record to the log. The record contains the IRL operations, the new inlist and outlist, and the partition id.

If the OR crashes during GC of partition p but nothing has been written to the disk and log yet, then GC can simply restart after recovery because no persistent state has been modified.

If the OR crashes after writing the partition but before writing the log record, the only information that is lost is the new outlist of p and the relevant `DecrCount` operations due to removal of inter-partition references by GC. (The installation of objects for p and the update of the inlist of p are not considered done before the end-of-GC record is written). This situation is still safe because GC only deletes inter-partition references. It does not introduce new ones. The discrepancies in the IRLs will be corrected when the partition is garbage collected next time.

5.6.3 IRL Processing

After processing the queued IRL operations, the IRL update thread writes out a log record that contains the a set of indices S of the log records that have been processed. (The record also contains the updated inlists, outlists, and possible IRL operations). The presence of this record supersedes the log records with indices contained in S. The IRL operations in the superseded log records will not be applied again by the recovery process.

5.7 Discussion

An issue related to the above optimized scheme is the potential large size of the queued IRL operations in the log. For example, the number of queued `InterRef` operations can become very large if the modified objects committed by clients contain a lot of object references. There are various ways to address this problem. First, the log can be compacted by removing redundant IRL operation entries. For example, if an `IncrCount` is followed by a `DecrCount` operation for the same inlist entry, both operations can be safely removed. Also, multiple `InterRef` operations from the *same* partition pointing to the *same* object can be reduced to a single `InterRef` operation. Second, the IRL operations can be represented in a more compact form. For instance, multiple `IncrCount` operations for the same inlist entry can be represented by a single `IncrCount` with a count that specifies the exact increment to be done. Alternatively, standard compression can be used to reduce the size of log records containing IRL operations. The records are decompressed prior to GC or IRL processing.

Another issue is that the scheme requires scanning the log to obtain a conservative inlist for GC and to process the queued IRL operations. In Thor, this does not raise a performance issue because the transaction log resides in main memory. For systems with disk-resident log, the queued IRL operations should be cached in memory (in a compact manner) to prevent reading log entries from disk.

Chapter 6

Conclusion

This thesis has presented the design of a garbage collection scheme for large object-oriented databases. This scheme uses a partitioned approach, which makes the garbage collection process efficient and scalable. The copying GC algorithm used in this scheme allows client operations to proceed with no extra overhead. Little synchronization is required between the OR garbage collector and the FEs. The recovery process is also simplified by using a copying algorithm.

We explained how the OR cooperates with the FEs to obtain part of the roots for garbage collection. We also described an incremental root generation algorithm that allows clients to proceed during root generation.

Our scheme uses inter-partition reference lists (IRLs) to maintain information necessary to perform independent collection of partitions. We addressed the performance issue related to the maintenance of IRLs. We devised an optimization that defers the IRL updates using the log and applies various techniques to reduce the number of I/Os required to perform the IRL updates.

Our work is done in the context of Thor and a prototype of our garbage collection scheme has been implemented. We believe our IRL update optimization is applicable to most object databases that use a partitioned GC approach.

6.1 Future Work

6.1.1 Performance Evaluation

It would be interesting to study different performance aspects of the scheme. First, the cost incurred by the maintenance of IRLs should be measured. Second, the performance of the garbage collector should be measured with no client, one client, and multiple clients running. Third, it would be interesting to vary the number of inter-partition references in the objects and see how it affects the cost of IRL maintenance. Finally, the performance of this scheme should be compared to other GC algorithms designed for object-oriented databases.

6.1.2 Inter-Partition Circular Garbage

Currently, the garbage collection scheme does not collect garbage with cycles of object references in different partitions. We believe this type of garbage is relatively infrequent. However, it is important to extend this scheme so that the storage of this type of garbage be reclaimed eventually or the uncollected garbage will accumulate over time. Several techniques have been considered.

- **Migration of objects**

  Circular garbage that spans across partitions can be collected if they are migrated into the same partition. The tricky part, however, is to identify potential garbage objects and decide where to move them. A migration scheme based on *distance* estimation of objects is described in [19]. The distance of an object is the number of internode references from any persistent root to that object. An object with a large distance is likely to be cyclic garbage. Although this work primarily aims at migration of objects between different servers, it can be easily adopted to use for object migration between partitions. However, this scheme does require the use of reference lists rather than reference counts for inlist entries.

- **Enlarging partitions**

  Instead of moving objects across partitions, it is possible to combine several partitions into one partition to contain potential circular garbage objects. Again, it is tricky to determine which partitions should be combined. It may be expensive to recompute the IRLs for the new combined partition. Recomputing the IRLs can be avoided if each inlist entry maintains a list of partitions with incoming references rather than just a count [8].

- **Complementary Tracing**

  It might be better to use a complementary algorithm based on global tracing [7, 11] because of its simplicity and completeness. The algorithm runs infrequently and should be designed to avoid excessive I/Os and paging.

6.1.3 Partition Selection Policies

An important aspect of partitioned GC algorithms is how to select a partition that contains the most garbage. Relatively little work has been done in this topic [5] and further study would be beneficial.

6.1.4 Clustering of Segments into Partitions

For performance reasons, it is important to cluster related objects into the same segment. When a database is divided into partitions, it is also important to group related segments into the same partition. The goal of this clustering is mainly to reduce the number of inter-partition references, which in turn reduces the cost of IRL updates and the chance of circular garbage. It would be valuable to study how this grouping can be determined. Related issues are how to determine the optimal partition size in relation to the segment size and whether reclustering of segments into partitions might help in collecting circular garbage.

Bibliography

[1] Atul Adya. Transaction management for mobile objects using optimistic concurrency control. Technical Report MIT/LCS/TR-626, Massachusetts Institute of Technology, January 1994.

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.

[3] Laurent Amsaleg, Michael Franklin, and Olivier Gruber. Efficient incremental garbage collection for client-server object database systems. In *Proceedings of the VLDB International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[4] Peter Bishop. Computer systems with a very large address space, and garbage collection. Technical Report MIT/LCS/TR-178, Massachusetts Institute of Technology, May 1977.

[5] Jonathan Cook, Alexander Wolf, and Benjamin Zorn. Partition selection policies in object database garbage collection. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 371–382, Mineapolis, Minnesota, May 1994.

[6] David Detlefs. Concurrent, atomic garbage collection. Technical Report CMU-CS-90-177, Carnegie Mellon University, October 1990.

[7] Edsger Dijkstra, Leslie Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[8] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proceedings of the ICDCS International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[9] Sanjay Ghemawat. The Modified Object Buffer: a storage management technique for object-oriented databases. Technical Report MIT/LCS/TR-666, Massachusetts Institute of Technology, September 1995.

[10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[11] John Hughes. *A Distributed Garbage Collection Algorithm*, pages 256–272. Number 201 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[12] Andrew Kirmse. Implementation of the two-phase commit protocol in Thor. Master's thesis, Massachusetts Institute of Technology, May 1995.

[13] Elliot Kolodner and William Weihl. *Atomic Incremental Garbage Collection*, pages 365–387. Number 637 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

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

[15] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew Myers, and Liuba Shrira. Safe and efficient sharing of persistant objects in Thor. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[16] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Distributed Object Management.* Morgan Kaufmann, 1994.

[17] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of the ACM SOSP Symposium on Operating Systems Principles*, pages 226–238, Pacific Grove, California, October 1991.

[18] Umesh Maheshwari and Barbara Liskov. Fault-tolerant distributed garbage collection in a client-server, object-oriented database. In *Proceedings of the PDIS International*

Conference on Parallel and Distributed Information Systems, pages 239–248, Austin, Texas, September 1994.

[19] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of the ACM PODC Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1995.

[20] Scott Nettles and James O'Toole. Real-time replication-based garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albequerque, NM, June 1993.

[21] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the ACM SOSP Symposium on Operating Systems Principles*, pages 161–174, Asheville, North Carolina, December 1993.

[22] James O'Toole and Liuba Shrira. Opportunistic log: Efficient reads in a reliable storage server. In *Proceedings of the Usenix OSDI Symposium on Operating Systems Design and Implementation*, Monterey, CA, 1994.

[23] Marc Shapiro, Olivier Gruber, and David Plainfosse. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

[24] Paul Wilson. *Uniprocessor Garbage Collection Techniques*, pages 1–42. Number 637 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[25] Voon-Fee Yong, Jeffrey Naughton, and Jie-Bing Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proceedings of the ICDE International Conference on Data Engineering*, pages 120–133, Houston, Texas, February 1994.