Efficient Garbage Collection for Large, Object-Oriented Databases

Tony C. Ng*
ng@lcs.mit.edu
MIT Laboratory for Computer Science

May 17, 1996

1 Introduction

Object-oriented databases (OODBs) provide persistent storage of objects with complex inter-relationships. 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, 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). The garbage collector has to minimize the number of disk I/Os and 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 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 [1], a distributed, client-server object-oriented database. Our scheme uses a partitioned approach [4]. 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. (2) The scheme is scalable because the amount of work done per GC is independent of the size of the database. (3) 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. (4) The scheme is fault-tolerant. Recovery is simple and fast even if the server crashes during GC.

Currently, our scheme needs to be augmented to collect objects with inter-partition reference cycles. We ignore the issue in this paper.

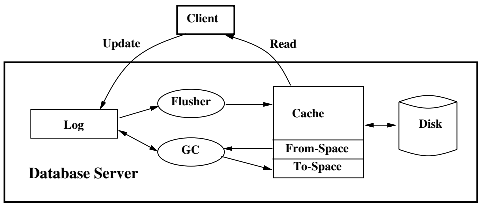

Figure 1: Thor Database Server

2 Thor Architecture

Figure 1 shows the system architecture of Thor, the database server on which our collection scheme is based. The storage of the database consists of a disk and a log. The disk stores the objects in the database and the log is used to hold recent updates from clients. The log resides in main-memory and stability is achieved by replication across multiple machines backed up by uninterruptible power supplies [2]. Reading from the log is the same as memory access and flushing a log record is equivalent to a network round-trip delay.

A client interacts with the server by reading and updating objects of the database. It reads the objects from the server cache. However, client updates are not written to the disk immediately. Instead, the modifications are appended to the log. In the background, the *flusher* thread moves the modified objects from the log to the disk. We refer to this as *object installation*.

*Tony Ng is advised by Barbara Liskov. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136. Address: Room 526, 545 Technology Square, Cambridge MA 02139. Web: http://www.pmg.lcs.mit.edu/∼ng

A portion of the server cache is used for garbage collection. The *from-space* is the partition currently being collected. During GC, the flusher is suspended so that the from-space will not be modified. The *to-space* is the new, empty partition created by the garbage collector. The garbage collector copies objects that are still used from the from-space to the to-space.

A replicating GC technique [3] is used to make GC concurrent. The collector only modifies the to-space. Clients can continue to fetch objects from the cache or from-space and continue to perform updates by appending modifications to the log. Very little synchronization is required between the collector and the clients. This scheme is fault-tolerant because if a server crashes during GC, the collector can simply discard the to-space and starts over. No recovery is required on the from-space because it is not modified.

3 Inter-Partition Reference Lists

To collect a partition independently, it is necessary to keep a list of incoming references from other partitions. This scheme uses inlists and outlists to keep track of inter-partition references. Each partition has an inlist and outlist, which are collectively referred to as the inter-partition reference lists (IRLs) in the rest of the paper.

The inlist is used to keep track of incoming references from other partitions. For each incoming reference, it maintains a count, which is the number of partitions containing this reference.

The outlist is a list of outgoing references to other partitions. It is not directly used for garbage collection. Instead, it allows efficient updates of the inlists.

There are two places where the IRLs must be updated. First, when the flusher moves modified objects from the log to disk, it has to add entries to the outlists and increment the inlist entries when it discovers new inter-partition references in the modified objects. Second, IRLs have to be updated after GC because inter-partition references might be removed as a result of the collection. This involves removing entries from the outlists and decrementing the inlist entries.

Note that IRLs are not updated immediately when a client performs updates. This means there is no extra overhead from IRL updates when a client modifies the objects in the database.

4 Deferred IRL Updates

The IRLs must be maintained persistently because recomputing them after server crash would involve scanning the entire database and make recovery process too slow and expensive. However, this means every IRL update is a potential disk read and write, which raises a performance issue.

To solve this problem, we have devised 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.

Three kinds of IRL operations are queued: IncrCount, DecrCount, and InterRef. 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.

By deferring the IRL operations, the flusher and collector incur minimal I/Os related to IRL updates. Only one stable write to the log is required at the end of object installation and GC to record the IRL operations. The log is compacted by removal of 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.

When the number of queued IRL operations in the log reaches a certain threshold, these operations are processed by a background thread. The cost of reading in and writing out the IRLs is amortized by performing the IRL updates in a batch. In addition, the IRLs are clustered together on disk to further reduce I/Os.

This technique is fault-tolerant. Since the deferred IRL updates are stored in the log, it will survive server crashes. In addition, modifications that required IRL updates are never removed from the log until the deferred IRL updates are written stably to the log. For example, the flusher will not remove the modified objects in the log until the IRL updates are logged stably.

References

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

[2] 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.

[3] 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.

[4] 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.