Fine-Grained Disk Space Accounting for Object-Oriented Databases

by

Marc LeBlanc

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1993

© Massachusetts Institute of Technology 1993. All rights reserved.

Author ..............................................................................
Department of Electrical Engineering and Computer Science
May 12, 1994

Certified by ............................................

Barbara Liskov
Professor of Computer Science
Thesis Supervisor

Certified by ............................................

Luis-Felipe Cabrera
Almaden Research Center
Thesis Supervisor

Accepted by ............................................

Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students
Fine-Grained Disk Space Accounting for Object-Oriented Databases

by

Marc LeBlanc

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

In this thesis, I explore the problem of determining the accountability of principals for their space usage in a persistent object store. I describe a system in which each principal pays for or “sponsors” the objects it wants to persist. Objects are paid for with disk quota or some other abstract resource, in proportion to the size of the object. I define the notion of “sponsorship links” as the mechanism through which a principal declares its sponsorship of an object, and discuss accounting system features needed to implement this mechanism. Several policies and algorithms for determining the total “charges” to a principal are presented and evaluated.

Thesis Supervisor: Barbara Liskov
Title: Professor of Computer Science

Thesis Supervisor: Luis-Felipe Cabrera
Title: Research Staff, Almaden Research Center
# Contents

1 Introduction ........................................... 5
  1.1 System Model ......................................... 5
  1.2 Accountability and Sponsorship ..................... 6
  1.3 Requirements ......................................... 6
  1.4 Bounds and Assumptions ............................ 7
  1.5 Overview ........................................... 7

2 A Framework for Accounting .......................... 9
  2.1 Establishing Sponsorship ............................ 9
    2.1.1 Rejected: Explicit Declaration of Sponsorship ... 9
    2.1.2 Data-Graph-Implicit Sponsorship ................. 10
    2.1.3 Example ......................................... 12
    2.1.4 Evaluation ....................................... 13
  2.2 Sponsorship of Newly-Created Objects ............ 13
    2.2.1 Volatile Objects ............................... 14
  2.3 Reclaiming Unsponsored Objects .................... 14
  2.4 Related Work ....................................... 14

3 Accounting Policies and Algorithms ................. 16
  3.1 Timing and Frequency of Accounting ............... 16
    3.1.1 Synchronizing with Garbage Collection .......... 17
  3.2 Sharing Policies .................................... 18
    3.2.1 Division of Costs ................................ 18
    3.2.2 All Pay Fully .................................. 19
    3.2.3 Hybrid Policies ................................ 19
  3.3 Accounting Algorithms .............................. 21
    3.3.1 Exhaustive Search .............................. 21
    3.3.2 Round-Robin Search ............................ 22
3.3.3 Single-Pass Accounting .................................................. 23
3.3.4 Formalization: The Complete Single-Pass Accounting Algorithm .......... 30
3.4 Evaluation ................................................................. 32

4 Conclusion ..................................................................... 34
4.1 Future Directions .......................................................... 34
  4.1.1 Programming Language Support ........................................ 34
  4.1.2 Fine-Grained Sponsorship ............................................... 35
  4.1.3 Parallelism ................................................................. 35
  4.1.4 Distribution ............................................................... 36


Chapter 1

Introduction

As information systems increase in size and complexity, more powerful tools are needed for managing and manipulating large stores of data. In response to this need, system developers are moving toward an approach that combines the high-volume storage management of database systems with the abstraction mechanisms of programming languages. This approach has led to the creation of persistent object stores, large disk-based databases whose structure conforms to the object-oriented programming model.

All objects in persistent object stores require system resources (i.e., space) to exist. Disk space is finite, and in large multi-user object stores there is a danger that disk space will not be shared fairly and appropriately by users, projects, and other accountable entities (collectively known as principals). However, by providing system administrators with a mechanism for monitoring—and perhaps limiting—the space usage of a particular principal, these circumstances can be avoided.

This thesis concerns itself with the problem of determining the accountability of each principal for the space usage in a persistent object store. It will present the concept of "sponsorship links" as a mechanism for determining the accountability of a principal for a particular object, and will discuss the implementation of this mechanism. It will discuss various policies and algorithms for computing the total space usage incurred by a particular principal, and provide an evaluation of these policies and algorithms. Only a limited amount of theoretical work has been done in this area. Discussion of this work appears in 2.4.

1.1 System Model

We will discuss the problems of accounting in terms of the following model of a persistent object store. The database is comprised of an amount of permanent storage, which is divided into a large number of objects. Objects may vary greatly in size, and may contain references to other objects. Each principal accesses objects by navigating the graph of objects and references that fans out from
its principal root object, the “primordial object” to which the principal always has access. A principal can modify the state of the database, creating new objects and mutating old ones, through atomic transactions. Newly created objects become persistent at the time the transaction that creates them commits. Memory in the object store is managed automatically by a garbage collector. The garbage collector is responsible for assuring that objects are destroyed when they are no longer in use by any principal. The destruction of these objects will happen in a manner consistent with the transaction mechanism, so that a particular transaction’s view of the data graph does not change as the transaction runs. Furthermore, we will assume that garbage collection happens in a “stop-the-world” fashion. Garbage collection occurs only when necessary, and all other computation is suspended while garbage collection occurs.

1.2 Accountability and Sponsorship

Our goal is to define a system where principals can be held accountable for the objects (and thereby the disk space) that they use. The accountability of principals for the objects they use is described in terms of sponsorship. A principal is said to “pay for” or sponsor the objects for which it is held accountable; the “cost” of an object is proportional to its size, and thus a principal’s total “debit” reflects the total amount of space it uses. A system administrator can limit a principal’s space usage by restricting the amount of “money” the principal has to spend on the sponsorship of objects. Sponsorship of an object guarantees its existence; the garbage collector is responsible for assuring that unsponsored objects are reclaimed. Ownership and sponsorship are not identical; a principal may sponsor an object it does not own, or own an object it does not sponsor. Furthermore, an object may have more than one sponsor, in which case each principal is “charged” for the object, according to some policy that assures that among the object’s sponsors the resources of the object have been paid for in full.

1.3 Requirements

The design of an accounting system is guided by a number of requirements and constraints, both pragmatic and academic. The most fundamental of these requirements is correctness; the total amount paid by principals must equal or exceed the total cost of all sponsored objects. This invariant guarantees that disk space never runs out, as long as the total amount of money or quota available for the sponsorship of objects does not exceed the cost of the total space available. In addition to being correct, we would also like an accounting system to be as accurate as possible. The ratio between the total amount paid by principals and the total cost for all sponsored objects should be as close to one as possible; ideally, the total cost of all sponsored objects matches the total space usage
exactly. However, when an object is sponsored by multiple principals, the total amount charged to the sponsoring principals might exceed the actual cost of the object. A policy that handles shared objects in this manner is still correct, but not accurate. The quota supply could be exhausted long before the supply of disk space. Whenever possible, we would like the system to be *fair*, to assure that all the principals sponsoring a given object will be charged equally for this object. In a fair system, the respective amounts charged to two different principals reflect the relative space usage of the principals. Finally, we would like our system to be *fine-grained*; that is, the system will be able to account for disk space on an object-by-object basis. The system should not compromise this level of accounting resolution.

While many of the requirements placed on the system represent genuine theoretical concerns, a number of pragmatic requirements also constrain the system. First among these is computational complexity. Because of their relatively low level of abstraction and their pervasiveness throughout the object system, the sponsorship mechanisms should not cause more than an incremental degradation of system performance. The accounting system needs to perform two operations: compute the total cost of all objects sponsored by a given principal, and garbage collect all unsponsored objects. Space and time costs for these functions should be as small as possible. Equally important to the computational performance is the necessity of a simple programming interface. The accounting system should require minimal (if any) extensions to the programming interface. The interface should allow programmers to assure the sponsorship of objects needed by their applications, but this should require only minimal effort on the programmer’s part.

### 1.4 Bounds and Assumptions

For the purposes of this thesis, the scope of the problem was limited by the bounds and assumptions described below. While some care in design and implementation has been taken to assure that these bounds may be removed in the future, the problems associated with these issues have not been thoroughly explored. We will assume, therefore, that:

- Only persistent objects need to be sponsored. Although many systems allow for volatile objects, they will not be considered because they may use a different accounting scheme.

- The system is not distributed, although distribution is prominent among possible extensions for future work.

### 1.5 Overview

In this work we describe a number of disk-space accounting systems in terms of sponsorship. Chapter 2 discusses a number of criteria for deciding whether a principal sponsors an object, and eventually
arrives at the notion of “sponsorship links.” It discusses some of the necessary mechanisms that are entailed by this notion. Chapter 3 discusses policies for resolving the cost of an object when it is sponsored by multiple principals, and presents algorithms that implement these policies. Finally, these algorithms and policies are evaluated in terms of their semantic and performance properties, and possible extensions to the accounting systems previously discussed are identified.
Chapter 2

A Framework for Accounting

In this chapter we examine the mechanisms by which a principal might be able to establish sponsorship of an object. Of these alternatives, the concept of “sponsorship links” proves most attractive. The chapter then discusses the features of the accounting system necessary to implement sponsorship links, for the purpose of charging for sponsored objects, reclaiming unsponsored ones, and assuring the sponsorship of newly-created objects.

2.1 Establishing Sponsorship

A resource accounting system must include a means of establishing whether an object is sponsored, and whether a given principal sponsors an object. The former is necessary so that unsponsored objects can be reclaimed, and the latter is necessary so that principals can be charged for the objects they sponsor. The programming interface must provide a means for an application to establish sponsorship of the data it uses on behalf of (and in the name of) the principal that invokes the application. Discussion of possible sponsorship mechanisms follows.

2.1.1 Rejected: Explicit Declaration of Sponsorship

Possibly the simplest mechanism to implement is one in which a principal (or the applications it uses) must explicitly declare its sponsorship of an object. The programming interface must provide three new functions: sponsor an object, revoke sponsorship of an object, and query the sponsorship extent of a principal. While simple to implement, this policy places an undue burden on applications programmers to assure that objects are properly sponsored when needed and sponsorship is properly revoked when not needed. This burden is analogous to the need for programmers to properly manage heap storage in programming languages (such as C and Pascal) that do not support

---

1 The sponsorship extent of a given principal is the set of objects that the principal sponsors.
garbage collection. Objects that are not properly sponsored (or freed too early) may vanish, leaving references dangling. Objects that never have their sponsorship revoked (or are never freed) may become unreachable, thereby wasting space (or quota). Because of the responsibility it requires of programmers, this mechanism is not a good choice for use in a space accounting system.

2.1.2 Data-Graph-Implicit Sponsorship

Instead of requiring principals to declare their sponsorship of objects explicitly, the sponsorship of objects can depend on the structure of the data graph. Applications can then affect the sponsorship of objects implicitly by mutating the data graph. If necessary, an application can modify the data graph with the explicit purpose of changing the sponsorship status of an object. The programming interface may have extensions to allow programmers to affect the interpretation of the data graph for the purposes of sponsorship.

Reachability as Sponsorship: Pros and Cons

Every principal has a root reference set, a set of references that define the root of the data graph as perceived by that principal. A principal can reach any object that it can reference transitively from its root reference set. A principal can access and manipulate only those objects it can reach. Objects that a principal cannot reach can be considered non-existent by that principal. One possible data-implicit sponsorship mechanism is to equate sponsorship with reachability. This is to say that a principal sponsors each and every object it can reach. The advantage to this policy is that it requires no extensions to the programming interface whatsoever; sponsorship is completely transparent to the application. Furthermore, unsponsored objects are necessarily garbage (unreachable by any principal) and, thus, a standard garbage collector can be used to reclaim unsponsored objects. However, this policy has a few notable shortcomings. Since an object sponsors each and every object it reaches, no principal can access an object it does not sponsor. Likewise, because of the transitivity of sponsorship, a principal could, with a single reference, unwittingly sponsor many more objects than it needs to manipulate. Furthermore, in a system that provides fine-grained access control of the data graph, a principal might reach (and thereby sponsor) an object to which it has no access.

Solution: Sponsorship Links

To overcome the shortcomings of equating sponsorship with reachability while preserving the desirable semantics of this mechanism, we divide references into two different strengths: strong and

\[\text{Assuming the programming interface provides a mechanism for sifting through a principal's sponsorship extent, a principal could find and revoke sponsorship for all objects it cannot reach (i.e., can reach only through the sponsorship extent). This, again, places an undue burden on the programmer.}\]
weak references. The strength of a reference is determined by the programmer. Sponsorship is then defined as strong-reachability; that is, a principal sponsors all objects it can reach through strong references exclusively. Thus, sponsorship is transitive across strong links. Because of this definition, strong references are also referred to as sponsorship links. Weak references are not considered part of the data graph for the purposes of sponsorship (or for the purposes of garbage collection). A weak reference does not provide sponsorship, and will dangle if the referenced object is reclaimed. At the same time, objects can use weak references to gain access to other objects without necessarily sponsoring them.

The sponsorship link mechanism requires an underlying system which is computationally more complex than the “explicit declaration” system described in 2.1.1. The accounting system must scan the data graph to determine the sponsorship status of an object. However, like with most garbage-collected systems, sponsorship links relieve the programmer of much responsibility. The use of sponsorship links allows applications to modify the sponsorship state of the data without violating abstraction barriers. Since an abstraction cannot affect sponsorship links that are not part of its view of the data graph (i.e., are outside its abstraction barrier), applications programmers need not worry about “malicious” abstractions causing sweeping changes in the sponsorship state of the data graph.

In this work we use sponsorship links as the mechanism for establishing and determining the sponsorship of objects.

Strong Instance Variables: Simplifying the Programming Interface

The programming interface must provide some facility for declaring the strength of a reference. The existence of a sponsorship link from one object to another implies a relationship of dependence; the referencing object cannot exist without the referenced object. When sponsorship links are put to proper use, typical objects will strongly reference (directly or transitively) the objects on which their functionality depends, and reference weakly those objects whose functions are useful but not vital. Thus, the strength of a reference to an object is (hopefully) dependent on the role that the referencree plays in the referencing object’s function. Since an object determines the roles of its subordinates according to the instance variables that hold references to them, the strength of an outgoing reference can be determined the same way. The strength of a reference will actually be a property of the instance variable that holds it, and the programming interface would require the programmer to specify which instance variables hold strong references, rather than the strengths of

---

3 Weak pointers (pointers that are ignored by the garbage collector) have appeared in various systems throughout the years. According to Mark Miller, co-author of [Drexler 88], weak references were used in the dialect of LISP known as T. Jonathan Rees, one of the T implementors, notes that weak pointers appeared as MAKNUM and MUNKAM in Multics Maclisp. MAKNUM maps an object to an integer, and MUNKAM inverts this mapping, provided the target object has not been reclaimed.
the references themselves. This provides a substantial simplification of the programming interface. Sponsoring instance variables are declared at object implementation time, as part of the semantics of the object.

This programming interface is not sufficient to allow programmers to assure the sponsorship of objects. Some mechanism must also be included to allow a principal to sponsor an object directly. That is, the programmer needs a way of modifying a principal's sponsorship root to strongly-reference new objects.

The existence of weak links, unlike the "reachability as sponsorship" policy (see 2.1.2), requires the cooperation of programmers to guarantee the correct sponsorship properties for the objects they implement. However, because strong references provide the sponsorship (i.e., persistence) semantics with which programmers are already familiar, and because holding multiple sponsorship links to an object does the principal no harm, strong references will probably be much more common than weak ones. Weak references are more useful for applications that cause the same data to be shared by multiple principals; one principal can hold a weak reference to objects sponsored by another principal. In the limiting case where all references are sponsorship links, the system behaves identically to the "reachability as sponsorship" system. Therefore, it is appropriate to specify that all instance variables of undeclared strength are sponsorship links by default, and only weak instance variables need to be declared explicitly. This further simplifies the programming interface; weak references become an "advanced feature" of the object definition language.

2.1.3 Example

Consider an indexing service that stores a mapping between key strings and documents. Clients of this service can add documents under the keys they specify, and can look up a document or documents based on a specified key. In order to maintain the state needed for these functions, the service object may create subordinate objects. Sponsors of this service should be held accountable for all of its component objects, but not for documents indexed by the service (which can be referenced weakly). Clients must continue to sponsor any documents in the service that they do not want reclaimed.

Figure 2-1 shows partial code for a binary tree implementation of this service, in which the internal nodes are sponsored but the documents are not. The code separates the definition into state and methods (typical for abstract data type languages), but also separates the state further into strong (instance variables holding strong references or "sponsorship links") and weak (instance variables holding weak references). Instance variables holding atomic data types (i.e., non-references) do not behave differently when declared under strong than under weak, because they are physically

---

\[4\]This is not entirely true in the object-weight accounting policy, where each strong link incurs a fraction of the referenced object's cost (see 3.2.3).
implementation indexnode1 /* implementation #1 of the indexnode type */
    strong
    left, right: indexnode; /* Child nodes, these are subparts of
    the index tree */
    key: string; /* Key for this node */

weak
    val: document; /* document stored under this key, NOT a subpart */

methods /* insert, delete, lookup, etc. */
(...method definitions...)
end

Figure 2-1: Type Declaration for a Binary Tree Implementation of an Indexing Service

The interface requires only one extra token to be typed by the programmer at implementation time. It does, however, require all object implementors to concern themselves with the semantics of reference strength. Perhaps the language will allow for a default programming interface in which reference strength is transparent, possibly by making inferences about the strength of instance variables. This thesis does not concern itself with investigating this possibility, leaving it as a direction for future work.

2.1.4 Evaluation

Given our system model in which objects are necessarily reclaimed automatically by a garbage collector, strong and weak links are the most appropriate mechanism for the establishment of sponsorship. This mechanism combines the flexibility of an explicit declaration system with the localized semantics of a reachability-based system. Different assumptions about the system model might suggest the use of a different mechanism. For example, in a system model where all object destruction is explicit, it would be appropriate for sponsorship to be explicit as well.

2.2 Sponsorship of Newly-Created Objects

Some care must be taken to assure the sponsorship of newly-created objects. Often enough, a principal will create an object through the invocation of an object creation service—an object whose purpose is to create other objects. The creation service would create the new object, perhaps perform some initialization of that object, and return a reference to it. When this occurs, there is some danger that, before a reference to the object can be secured in a strong link, the accounting mechanism will reclaim it for lack of sponsorship. The object creation service could establish a
strong link to it as soon as it is created, thus assuring its sponsorship. However, this requires some protocol by which the creation service can hand over the new object to the client. The service does not want to sponsor the object forever (or perhaps at all), and the client needs to assure the objects existence long enough to establish a sponsorship link.

2.2.1 Volatile Objects

To solve the problem of sponsorship of newly-created objects, we declare all objects to be volatile at the time they are created. Volatile objects do not need sponsorship, but their existence is guaranteed only to the end of the transaction that created them. When the transaction commits, all volatile objects that are strong-reachable from some persistent object are made persistent, or solidified. Volatile objects that are weak-reachable (but not strong-reachable) from a persistent object are not solidified. They force the transaction into an error state, in which it cannot commit. The strict enforcement of sponsorship or complete unreachability for volatile objects enables programmers to more easily find errors in which the sponsorship of a newly-created object was accidentally forgotten.

2.3 Reclaiming Unsponsored Objects

With few algorithmic changes, a standard garbage collection algorithm can be modified to reclaim unsponsored objects. The garbage collector must ignore weak references when determining the “reachability” of an object. Weak links, however, must be properly “cauterized” in the case where they dangle, to prevent memory faults and mistaken identities. When garbage collection occurs in the middle of a transaction, some care must be taken to assure that objects which have already been accessed by the transaction are not reclaimed, since the transaction might cause the current principal to sponsor those objects. This can be accomplished by adding the set of actively used objects to the current principal’s sponsorship root.

2.4 Related Work

Although very few current applications concern themselves with fine-grained accounting of space resources, some theoretical work has been done in this area. [Drexler 88] describes an economic model for resource accounting, in which an object is charged “rent” for the space it occupies, and pays for the rent by charging “retainer fees” to its referencing objects. Principals are allotted an “income” with which they can pay retainer fees, and any object whose rent is supplied by some principal, directly or transitively, is guaranteed to exist. Garbage objects thus have no income, and are eventually reclaimed because of their poverty. The system administrator can adjust the income levels of principals to reflect the resource needs of users. This model provides a convenient mechanism
for garbage collecting distributed cycles; a cycle need not be detected, as it will eventually become impoverished and its components reclaimed. [Drexler 88] also proposes a more general version of reference strength, in which references are able to specify a bounded economic commitment to the maintenance of the referenced object. This is to say that such a reference will sponsor the referenced object so long as the cost does not exceed some threshold level.\(^5\)

\(^5\)See 4.1.2 for some discussion of this generalization as a future direction for this work.
Chapter 3

Accounting Policies and Algorithms

This chapter focuses on the actual process of accounting. It discusses the constraints on the timing and frequency of accounting. It then presents a number of policies for determining the cost charged to each of an object’s sponsors, and the algorithms capable of implementing them. Finally, the chapter will present and evaluate the set of accounting systems that these policies and algorithms can implement.

3.1 Timing and Frequency of Accounting

In designing policies and algorithms of the accounting system, it is necessary to consider the frequency with which accounting needs to be performed and how up-to-date the accounting information needs to be. An accounting system that updates frequently must perform each update quickly, whereas a system with only periodic updates need not be so performance-intensive. Ideally, the accounting system would reflect changes in the data graph as soon as they happen; whenever a transaction committed, the accounting system would determine whether the transaction made any changes to the data graph that affected the sponsorship status of objects, and adjust the relevant principals’ quotas in light of these changes. Unfortunately, our reachability-based definition of sponsorship does not lend itself easily to this instantaneous accounting method. Depending on the connectivity of the data graph, the modification of a single strong link could cause a principal to newly sponsor (or no longer sponsor) large numbers of objects. To compute a quota change for that principal, the accounting system would have to scan every object reachable from the modified reference, based on its value both before and after it was modified, to determine the change in the principal’s quota. This adds significant overhead to transaction commit time. Transactions
that normally run in constant time might take a full scan of the data graph to commit. Thus, reachability-based sponsorship makes up-to-the-minute accounting infeasible.

Given the impracticality of up-to-the-minute accounting, the system must resign itself to performing fairly infrequent accounting passes, in which charges to principals are updated based on changes to the data graph. The necessary frequency of accounting is largely dependent on the needs of the system. Some systems must have accounting information which is as accurate as possible, and will perform accounting at every practical opportunity. Other systems need to see only general trends in resource use, and can afford to perform an accounting pass as infrequently as daily or weekly. If accounting occurs particularly infrequently, the system may wish to take extra precautions to assure that space does not run out due to principals exceeding their quota in between accounting passes. One strategy for dealing with this is to require any principal that creates a new persistent object to provide that object with provisional sponsorship until the next accounting pass. The cost of the object would be deducted from the principal’s quota when the object is created (i.e., at commit time of the transaction that created the object). Another strategy is to perform “emergency accounting” whenever space usage in the object store crosses some critical threshold.

3.1.1 Synchronizing with Garbage Collection

Another important constraint on the timing of accounting is the synchronization invariant: The process of computing the total sponsorship debit accrued by each principal must synchronize with (i.e., occur at the same logical time as) a garbage collection.

Conceptually, the sponsorship of objects is implemented and enforced by two entities: the accounting system and the garbage collector. The accounting system is responsible for ensuring that each principal has paid for the objects it sponsors, and the garbage collector is responsible for ensuring that sponsored objects persist and that unsponsored objects are reclaimed when space is needed. For these systems to implement sponsorship properly, they must remain synchronized with one another. This is to say that accounting and garbage collection must be performed on identical snapshots of the data graph. Otherwise, an abusive principal could add hard links to objects just before the garbage collector finds them, and then remove them immediately after the garbage collector scans the object. The accounting system would never see these hard links, and thus would not charge for them. This would allow a principal to “juggle” many more objects than its quota permits it to sponsor. However, if every accounting pass occurs at the same logical instant as a garbage collection, a principal cannot cause accounting and garbage collection to work with different versions of the data graph; thus, “juggling” is impossible. This constraint is easy to preserve in a simple “stop the world” garbage collector; the system performs accounting while the “world is stopped” for garbage collection. Since “stop the world” garbage collection is typically invoked whenever a principal attempts to create an object for which there is insufficient space in the object
store, accounting at the same time as garbage collection will also enable the system to determine whether the current principal has enough quota to support the object being created. This invariant provides an important lower bound on the frequency of accounting. Since every accounting pass must include a garbage collection, accounting may not occur more frequently than garbage collection. Future work might consider the task of preserving the synchronization invariant in systems where garbage collection is interleaved with other transactions.

The synchronization invariant does not imply that every garbage collection must include an accounting pass. In the case where garbage collection occurs but accounting does not, accounting information will not reflect the deletion of newly-reclaimed objects. Thus, after an object has been reclaimed, its sponsors will still pay for it until the next accounting pass.

### 3.2 Sharing Policies

When more than one principal sponsors a given object, each sponsoring principal must be charged according to some policy that assures that among them, they have paid for the object in full. If the policy is constrained by fairness and correctness (see 1.3), then all sponsoring principals must be charged the same amount, which is minimally equal to the cost of the object divided by the number of sponsoring principals. Sharing policies are closely coupled with the accounting algorithms that implement them, though different variations on the same algorithm can implement different policies.

#### 3.2.1 Division of Costs

With regard to fairness, accuracy and correctness, the “division-of-cost” policy is the “perfect” solution. The cost of an object is divided equally among the principals that sponsor it. Thus, the total cost charged to a principal reflects its exact share of the total space usage. Since every object is paid for exactly by its sponsors, the total amount spent by principals reflects the exact amount of disk space used. Sharing is encouraged by virtue of the fact that sharing an object is less expensive than copying it.

One unfortunate consequence of this policy is its counterintuitive behavior. From a single principal’s perspective, traditional accounting schemes for filesystems adhere to a simple model: the user has a “virtual disk,” whose capacity is equal to the user’s quota. This virtual disk is “filled up” by the files that the user owns and, once it is full, the user has no more space for files left. A file exists in its entirety on a single user’s virtual disk, and that user pays the whole cost of the file. The division-of-cost policy does not adhere to this model; a user may pay some fraction of the cost of any object it sponsors, and that fraction may change from time to time based on the number of principals that share the object. This results in a certain “instability” in the cost of an object: If enough principals abandon an object leaving a single principal to sponsor it, that principal might
find that it has exceeded its quota without sponsoring any new objects.

3.2.2 All Pay Fully

Rather than dividing the cost of an object equally among its sponsors, the “all-pay-fully” policy charges the full cost of the object to each of the object’s sponsors. This policy is both fair and correct; two principals will both pay the same amount for a given object and every sponsored object is sure to be paid for in full (with some overkill in the case of sharing). However, sharing of objects between principals increases the amount of “overbilling,” and thus decreases accuracy. At worst, every principal could be charged for the entire cost of the database. Shared objects are no cheaper than copied objects, so there is no incentive for principals to save space by sharing. On the other hand, this policy does not suffer from the instability of the division-of-costs policy. The total charges to a principal will never change unless that principal sponsors new objects or revokes sponsorship of old ones.

3.2.3 Hybrid Policies

The “division-of-cost” and “all-pay-fully” policies represent the reasonable lower and upper bounds for fair policies. No policy which is fair and accurate can charge a principal less than $\frac{s}{p}$, where $s$ is the size of the object and $p$ is the number of sponsors. To account for the full cost of an object, a policy that charges one principal less than $\frac{s}{p}$ must charge another principal more, thus violating fairness. Likewise, no correct policy need charge a principal more than $s$ for an object, since charging $s$ is enough to assure that the object is fully paid for. This section describes other policies that lie between these extremes.

Sharing-Based Cost Scaling

The accounting system can compensate for the inaccuracy of the all-pay-fully policy by scaling the cost of every object to reflect the amount of overbilling. That is, the accounting system computes the sharing ratio, the ratio between the total number of size-units charged to principals and the actual amount of disk space used, and scales the cost of all objects down by that factor. As a result, the total cost paid for objects is now equal to the amount of disk space used by the objects. This scaling technique eliminates the inaccuracy of overbilling, but also introduces an instability similar to that seen in the division-of-cost policy: a principal’s total charges are affected by the actions of other principals, and a decrease in sharing can create a sudden rise in every principal’s costs. However, the instability is not as great as with the division-of-cost policy, since all quota changes due to increased or decreased sharing are averaged over every principal. By reducing all costs whenever sharing increases, scaling encourages the “community at large” to share objects. It
Figure 3-1: The left half of the data graph is sponsored by one principal, while the right half is sponsored by five. If the two halves of the data graph are equal in size, then the total sharing ratio for an all-pay-fully accounting scheme would be 3 : 1. In a scaling system, the left hand principal would be charged \( \frac{1}{3} \) of the cost of the objects it sponsors, even though it shares no objects.

It does not, however, encourage individual principals. An individual principal can capitalize on other principals’ frugality, without sharing any objects itself. For example, consider a data graph with two disjoint halves of equal size. A single principal \( P \) sponsors the first half of the data graph exclusively, while all other principals share the second half. Principal \( P \) will pay the same amount for the first half of the data graph as any other principal will pay for the second, even though \( P \) uses much more disk space than any other principal (see Figure 3-1).

The scaling system can reduce instability by performing a more time-dependant analysis of sharing in the system. Rather than pricing objects based on the current sharing ratio, the system could analyse the sharing ratio over a period of time, in an attempt to predict trends and recognize temporary anomalies. Such a system must distinguish between the sharing ratio, which reflects the current state of the system, and the pricing ratio, which is used in scaling the cost of an object. Changes to the pricing ratio will lag behind changes to the sharing ratio, and thus temporary spikes in the sharing ratio will be ignored. In one such system, the pricing ratio might converge to the sharing ratio along a decaying exponential. Another might try to fit the changes in the sharing ratio to a polynomial curve, and adjust the pricing ratio according to that curve. Even more sophisticated analysis is possible, and a study of scaling strategies might be an appropriate direction for future work.

**Object Weight Analysis**

Rather than dividing the cost of an object equally among its sponsors, the accounting system might bias the size of a sponsor’s share based on the actual paths of references from the sponsor to the object. Sponsors with many bifurcating paths to an object might pay more than sponsors with few.
Figure 3-2: An Example of Object Weight Analysis. Three principals, A, B and C sponsor objects X and Y through different paths. X has a “weight” of eight, whereas Y has a “weight” of two. The load of X is divided equally among its two references, thus A bears a load of four, and Y’s load is increased by four. In turn, Y’s total load of six is divided equally between B and C. Between these three principals X and Y are paid for exactly.

Object weight analysis has such a property. This policy describes the cost incurred by principals in terms of a physical metaphor: the objects in the data graph are viewed as weights, with mass in proportion to their size. References are “ropes” that hold up the objects, and principal roots are “anchor points” that can take an amount of load up to their quota. Each principal is charged an amount equal to the total load on its principal root. Since the entire system of weights is suspended statically by the principals, there must be exactly enough “force” distributed among the principals to account for every non-garbage object. Thus the policy is perfectly accurate. Garbage objects have no support from principals, and thus fall into oblivion. This policy is not necessarily fair, because a particular principal’s contribution to an object’s cost depends on its paths to that object, and on who shares those paths (see Figure 3-2). Object weight analysis shares many of the properties of the division-of-cost policy: it is accurate and encourages sharing, but possesses the same instability.

3.3 Accounting Algorithms

In this section we will survey a number of algorithms, and discuss the policies they are capable of implementing.

3.3.1 Exhaustive Search

Perhaps the most straightforward (and costly) algorithm for accounting is an exhaustive search, in which the entire data graph is searched once for each principal. The search begins at the “first” principal’s root object, and continues across strong links until every object that is strongly-reachable from that root is found. The search then begins again with a different principal. Whenever the search
traverses an object, it performs some accounting to reflect the fact that the current principal sponsors
the object being traversed. Using exhaustive search, we can implement any of the accounting policies
described in the first section.

In general, exhaustive search proves to be too expensive to be run very frequently. In the worst
case, it grows in time as $\Theta(PS)$, where $S$ is the total size of the object space and $P$ is the number
of principals. As sharing increases, so does the time required to perform accounting. An exhaustive
search does the work of garbage collection many times over, and thus should be performed much
less frequently than every garbage collection cycle. On the other hand, exhaustive searches lend
themselves quite well to parallelization, as every principal's search can be performed independently.

### 3.3.2 Round-Robin Search

The round-robin search is a variant on the exhaustive search, based on the fact that accounting
does all of the work of garbage collection (and more), and that accounting must preserve the syn-
chronization invariant (see 3.1.1). Since exhaustive-search accounting requires more computation
time than garbage collection, a full accounting pass cannot be performed every time the world is
stopped for garbage collection. Instead, the round-robin search performs an accounting pass for a
single principal during each garbage collection cycle. When the world is stopped for garbage collec-
tion, the system searches the data graph from a single principal, performing accounting as if that
principal were the "current principal" in the exhaustive-search algorithm, updating table entries or
adding costs as appropriate to the sharing policy. The search collects garbage at the same time,
marking or copying each object traversed so that its space will not be reclaimed. Once the first
principal is finished, the search continues from another principal's root. However, accounting is not
performed when searching from any principal root other than the first one, and objects that have
been previously traversed are not traversed again. Every garbage collection cycle begins with a
different principal's root, until accounting has been performed once for each principal. The cycle
then repeats, updating a single principal's charges every garbage collection, and thereby performing
the equivalent of a full exhaustive search every $P$ cycles, where $P$ is the number of principals. Thus,
the round-robin search is simply a formal schedule for the execution of an exhaustive search, in
principal-sized pieces, during each garbage collection. Because of this, the round robin search is ca-
ble of implementing any policy that exhaustive search is. However, since the round-robin search is
an "incremental" implementation of the exhaustive search, some policies may require a large amount
of accounting data to be preserved from one garbage collection to the next. For example, in order
for the division-of-cost policy to be implemented in a round-robin fashion, it would be necessary to
maintain a table mapping each object to the number (or, perhaps, the set) of principals that sponsor
it. The all-pay-fully policy, however, requires no such state to be preserved.
Optimizing for Disjoint Principals

When two or more principals sponsor disjoint subsets of the data graph, the round-robin search can be modified to perform accounting on such disjoint principals during the same garbage collection cycle, exceeding the normal limit of one principal per cycle. The search begins as before, with accounting being performed on the first principal searched. The algorithm performs accounting on the second and subsequent principals as well, in the hope that some of these principals will be disjoint with every already-searched principal. As soon as an object that had already been traversed by a previous principal is discovered, the algorithm backs out all the accounting performed for the current principal.\(^1\) If an entire principal’s search is concluded without discovering any previously-traversed objects, that principal is disjoint, and the accounting information is valid. Note that this will not allow all disjoint principals to be computed all the time. Suppose some principal $P$ shares objects with every other principal. If $P$ is searched first in a cycle, it will prevent accounting for all other principals. By the same token, if any principal is disjoint from all other principals, its costs will be recomputed every garbage collection cycle. Perhaps the algorithm could be further augmented to record information on disjointness relationships, which could then be used to produce a search order of principals that would maximize disjointness. Such an augmentation might be an appropriate area for future work.

3.3.3 Single-Pass Accounting

The single-pass algorithm computes accurate object-weight accounting for all principals in a single scan of the data graph, sacrificing some fairness and space overhead. Because the algorithm makes exactly one pass on the data graph, it can garbage collect at the same time as accounting.

The algorithm works by maintaining a mapping of each object to its “cost.” In this case, the “cost” of an object is its size (i.e., its “weight”), plus the “salaries” it pays to the objects it references. The “salary” of an object is always proportional to its “cost” (and thus, the algorithm is necessarily recursive). In the object-weight metaphor, the “salary” of an object reflects the “load” an object places on its referencers. In a bottom-up fashion, each object adds its load to the objects that reference it, which in turn add their load to their referencers, and so on until costs propagate all the way to the principal roots.

Basic Overview

The simplest case of this algorithm is one in which every object (other than a principal root) is strongly-referenced by exactly one other object, and thus each principal’s sponsorship extent is a

---

\(^1\) In a copy-compaction garbage collector such as the one described in [Baker 78], whether an object was traversed by a previous principal can be detected very simply: such an object will have been copied into new-space earlier than the current principal’s root object, and thus its address will be lower.
Tree-compute-cost(Object)
    Cost[Object] := Size[Object]
    For each Child ∈ Children[Object]
        Tree-compute-cost(Child)
    End
End

Figure 3-3: The Single-Pass Accounting Algorithm for Trees. After Tree-compute-cost is called on a particular object, Cost[Object] is set to the total cost of that object and all of its subordinates. Thus, accounting can be performed by running Tree-compute-cost on each principal root.

tree. In this case, the leaves of the tree have no subordinates, and thus pay no salaries; each leaf’s cost is exactly equal to its size. When the leaves have charged their salaries to their parents, the cost of every object that references only leaves is known. The process continues; as soon as the cost of a node is known (every object it references has charged it a salary), that object charges its salary to its parent. The algorithm halts once the cost of every principal root is known. The amount charged to each principal is, thus, the cost of the principal root. It is easy to see that every principal is charged a cost equal to the total size of all objects in its tree. The costs can be computed using a simple depth-first search of each tree. As the search unwinds across a node, the cost of the node is added to its parent; the costs of the node and all its children are known. The search is formalized in figure 3-3. It is easy to see that the algorithm has a runtime of $\Theta(V + E)$, where $V$ is the number of objects (vertices) and $E$ is the number of references (edges) in the data graph. Since references have constant size and are contained inside objects $\Theta(V + E)$ is bounded linearly in the size of the data graph.

Most data graphs are not trees, and this simple depth-first search will need to be augmented to work for arbitrary data graphs. Let us first consider the results when Tree-compute-cost is applied to a directed acyclic graph (dag). Objects may be referenced by more than one object and principals can now share objects. In this case, an object’s salary will be charged to each of its parents. Thus, a principal will pay the full cost of the object once for each path of sponsorship links from the principal to the object. This scheme is technically correct because a principal is charged at least once for every object it sponsors. However, the algorithm is neither accurate nor fair because the number of paths to an object can grow exponentially in the total size of the data graph, or worse. Two principals sponsoring disjoint, but equally-sized, subsets of the data graph can be charged widely different amounts.

To improve the accuracy of our algorithm, we will need to redefine the salary of an object when

---

2 Consider the data graph that is a linear chain of strong links. From each object add a strong link to every object that comes later in the chain. In the resulting graph, there are $2^n$ paths to each object, where $n$ is the object’s place in the chain. Since data graphs are multigraphs, this is not an upper bound.
an object is referenced more than once. One way to do this is the following. Let us return for a moment to the object-weight metaphor. An object referenced multiply has more than one “rope” to bear its load. Each of these ropes would bear the load equally, thus, the load on a particular rope would be the total load of the object divided by the total number of ropes. We can therefore redefine the “salary” of an object as its cost divided by its reference count, the number of objects that strongly reference it. In the case of the tree, all objects have a reference count of one, and thus this definition of salary still holds in this case. As discussed in 3.2.3, object-weight accounting does not necessarily charge each principal the same amount for an object, depending on the structure of the data graph. It is important to note, however, that a principal never pays more than the full cost of an object.

This new definition of salary raises a very important question: how is the reference count of an object known? Reference counts can be computed by one of two methods. First, the accounting system can make a separate pass over the data graph to compute the reference counts of every object in it. This first pass also runs in $O(V + E)$, and thus does not increase the asymptotic bound of the accounting process as a whole. However, this method changes the accounting algorithm from a single-pass algorithm to a two-pass algorithm. The second method is to store a reference count for each object at all times. Reference counts can be incrementally updated by scanning the set of modified objects at transaction commit time, incrementing and decrementing reference counts based on the (respective) adding and deleting of references to objects. This second method adds extra time overhead to transaction commit, as well as space overhead of $\Theta(V)$.

The tree-accounting algorithm can be made to work for a dag simply by modifying the definition of salary to factor in the object’s reference count. Aside from the definition of salary, the algorithm requires only one other change: since an object can now be reached through multiple search paths, a naive depth-first search may do extra work as it scans an object once for each path to the object. Thus, the tree-accounting algorithm must also be augmented to keep track of the objects it has already seen, in order to assure that each node is traversed only once. The code for this algorithm is shown in figure 3-4.

**Detecting Cycles**

Finally, we must augment our algorithm to handle cycles in the data graph. Previously, we defined cost recursively as,

$$\text{Cost}(x) = \text{Size}(x) + \sum_{c \in S(x)} \frac{\text{Cost}(c) / \text{refcount}(c)}{\text{refcount}(c)},$$

where $S(x)$ is the set of objects that are strongly referenced by object $x$ and refcount$(x)$ is the number of strong references that point at object $x$. In a cyclic graph, however, this recursion makes no sense; the cost of an object would be defined in terms of itself. In order to augment our algorithm,
Dag-compute-cost(Object)
    if Status[Object] = done then return
    Cost[Object] := Size[Object]
    For each Child ∈ Children[Object]
        Dag-compute-cost(Child)
    End
    Status[Object] := done
End

Figure 3-4: The Single-Pass Accounting Algorithm for Dags

we must first augment our notion of cost. This can be accomplished by constructing from the data graph \( G_D \) a new graph \( M(G_D) \) of the maximal strongly-connected components in \( G_D \). Each node in \( M(G_D) \) represents a maximal strongly-connected component in \( G_D \), and each reference \( A \rightarrow B \) in \( M(G_D) \) represents a single reference in \( G_D \) between some object in component \( A \) of \( G_D \) and a different object in a different component \( B \) in \( G_D \) (see figure 3-5). References in \( G_D \) that are internal to a strongly-connected component have no representation in \( M(G_D) \). However, it is possible for \( M(G_D) \) to have multiple references between the same two source and target nodes if they represent different references in \( G_D \). It should be clear that, by definition, \( M(G_D) \) has no strongly-connected components and is thus a dag. We can then define the size of a node in \( M(G_D) \) to be equal to the total size of all the nodes that make up its component in \( G_D \), and compute the cost of each node in \( M(G_D) \) using the reference-count-based dag algorithm. To extend the definition of cost to include cyclic data graphs, we define the cost of an object in \( G_D \) to be equal to the cost of the node in \( M(G_D) \) that corresponds to its strongly-connected component. This definition properly generalizes the recursive definitions for dags: in the degenerate case where \( G_D \) is a dag, \( G_D = M(G_D) \) and the recursive formula holds. This treatment of cycles makes a great deal of semantic sense. A strongly-connected component lives and dies as a single unit as far as the garbage collector is concerned, and all of a strongly-connected component’s internal sponsorship links are of equal value for “holding the component together.” It is, therefore, appropriate for principals to treat strongly-connected components as single units for the purpose of sponsorship.

Our algorithm needs to be able to detect cycles, and treat them as a single node for the purpose of accounting. In order to accomplish this, it will need to add more accounting data for each object. The acyclic algorithm keeps track of the state of each object: whether it is known or unknown. The cycle-detecting algorithm will know whether an object is unseen, seen or done. Unseen means that

---

3A strongly-connected component is a subgraph that contains a path from every node to every other node. Thus, every node is contained some cycle that passes through the strongly-connected component. A maximal strongly-connected component is one that is maximal in size; there is no larger strongly-connected component that contains it.
Figure 3-5: A Sample Data Graph. Figure (a) shows a graph $G_D$. Dotted lines indicate the maximal strongly connected components. Figure (b) shows $M(G_D)$. Note that there are two references from $DGFH$ to $CKEJ$ in $M(G_D)$, one for $G \to E$ and one for $D \to C$ in $G_D$.

The algorithm has not yet found the object. $Seen$ means the object has been found, but the search has not yet unwound back across it. When the search unwinds past an object, it is marked $done$.

The cycle-detector finds strongly-connected components by working from the observation that, since the search is depth-first, all of the nodes whose status is currently $seen$ form a simple path from the root of the search to the object currently being searched (see Figure 3-6). Thus, there is a path from every $seen$ node to the current node. Furthermore, the path formed by these $seen$ nodes is the currently active branch of the depth-first search tree. So any time the current node is found to reference another $seen$ node, these two nodes must be connected in a cycle. The contents of this cycle can then be found by unwinding the depth-first search from one node to the other.

Let us first consider the example of a simple cycle. Suppose the search in Figure 3-6 reveals a reference from $A$ to $B$. Once this reference is found, we will note the existence of a cycle by declaring $B$ to be the cycle's "representative." For every object $n$, we will keep a reference $rep[n]$ to the representative of the cycle to which it belongs. Initially, all objects are considered to be members of their own singleton cycle (whether or not this is actually true), and are thus their own representatives. Since we have assumed that the cycle is simple, we know that $B$ is the earliest member of the cycle to be found by the depth-first search, making $B$ a good choice for the cycle's representative. When $B$ has been marked $done$, we know that $B$'s cycle has been completely traced. When the representative of a cycle is marked $done$, we say that the cycle closes. For the sake of simplicity, let us assume that $A$ has no more outgoing references, and the search will begin to unwind immediately. When the search unwinds to $C$, we know that $A$ and $C$ are part of a cycle by virtue of
Figure 3-6: A Depth-First Search. Arrows and numbers show the order in which nodes are visited, and the paths on which they are reached. White nodes are unseen, grey nodes are seen, black nodes are done. Note that at the time that object A has been visited, all grey nodes have a path to A. If a reference were found from A to another grey node such as B, this would imply the existence of a cycle including A, B and C.

the fact that \( \text{rep}[A] \neq A \), and so we set \( \text{rep}[C] := \text{rep}[A] \) also. As we unwind, for each object \( X \) with child \( Y \) such that \( \text{rep}[Y] \neq Y \), we will set \( \text{rep}[X] := \text{rep}[Y] \), until eventually the search unwinds to \( \text{rep}[Y] \). At the same time, the Cost\([Y]\) is added to Cost\([X]\). In our example, the representative of each grey node will be set to B, until the search unwinds past B, the representative itself. When the search is finished, the set of nodes whose representative is B will be members of the cycle containing B. Furthermore, Cost\([B]\) will be the total cost of all members of that cycle.

Occasionally, a cycle will be found between two strongly-connected components \( P \) and \( Q \), and it will be necessary to merge them into a single component \( P \cup Q \). We will accomplish this merge by choosing the representative of, say, \( P \) as the representative of the \( P \cup Q \), and then setting \( \text{rep}[q] := p \), where \( p \) and \( q \) are the representatives of \( P \) and \( Q \), respectively. Note that, before the union of \( P \) and \( Q \) took place, every element \( n \) of \( Q \) had \( \text{rep}[n] = q \), and the union only modifies \( \text{rep}[q] \), leaving the representative of every other element of \( Q \) unchanged, and thus their representative fields obsolete. We should note, however, that we can still find the representative of \( n \in Q \) by following the chain of representative pointers until we find an object that represents itself. In our example, \( \text{rep}[n] = q \) and \( \text{rep}[q] = p \) and \( \text{rep}[p] = p \), so the true representative of \( n \) is \( p \). However, the algorithm does not need to query the true representative of a particular object. Rather, it needs to know when a particular object is a representative, so that it can perform appropriate accounting functions when a strongly-connected component closes. When we perform a merge on two components, we will always choose the earlier representative of the two to be the representative of the union. The earlier representative is the one first found and marked seen by the search.\(^4\)

\(^4\)Determining which of two objects is earlier is quite simple if we are performing copy-compaction garbage collection
a matter of replacing every assignment $\text{rep}[X] := Y$ with $\text{rep}[X] := \text{earlier}(\text{rep}[X], Y)$. This method of detecting cycles is not new. [Melhorn 84] presents a cycle detection algorithm very similar to this one, complete with a correctness proof.

When a component closes, the cost of the component is propagated back to every object in the component. This can be done by keeping track of all the references that are internal to the component, and searching from the representative along internal references until every object in the component is found. The cost of this propagation is linear in the size of the component. Since every object belongs to one component, the net cost of all propagation is linear in the size of the data graph. At the same time the cost is propagated, the total number of external references — references from outside the component — are computed and also propagated. This number is used as the reference count for any object in the component when computing salary. Once this is done, every element of the component has the cost and reference count of the component as a whole. Thus, accounting can proceed as if the component were a single node.

\[\text{at the same time as accounting (see [Baker 78]); we copy an object into new-space the same time as we would mark it seen, and earlier objects appear physically earlier in new-space.}\]
3.3.4 Formalization: The Complete Single-Pass Accounting Algorithm

With all the tools for cycle detection and accounting in place, we can now formalize the algorithm.

State

The following table shows the accounting information that will be stored for each object \( N \).

<table>
<thead>
<tr>
<th>Notation</th>
<th>Initial Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status(N)</td>
<td>unseen</td>
<td>Value is one of: unseen, seen, done, or traced. Unseen, seen, and done are described above in 3.3.3. When the search unwinds across an object in a not-yet-closed component, the object is marked traced rather than done.</td>
</tr>
<tr>
<td>Cost(N)</td>
<td>SizeOf(N)</td>
<td>The cost of object ( N ), including salaries paid out to subordinates, and including the cost of all other objects in ( N )'s component.</td>
</tr>
<tr>
<td>Rep(N)</td>
<td>( N )</td>
<td>A pointer that leads to the representative of ( N )'s component. If ( \text{Rep}[N] = N ), then ( N ) is the representative of its component.</td>
</tr>
<tr>
<td>Internals(N)</td>
<td>( \phi )</td>
<td>The subset of ( N )'s strong references that are internal to ( N )'s component. Note that this requires only a single bit for each reference, which is much smaller than the reference itself.</td>
</tr>
<tr>
<td>Fanin(N)</td>
<td>Refcount(N)</td>
<td>The number of references in ( N )'s component that are external to ( N ), that is, the reference count of ( N )'s component when seen as a single node.</td>
</tr>
</tbody>
</table>

The accounting data for an object is assumed to be initialized the first time it is needed.

Accounting Procedure

Here is the accounting process, step by step.

Compute-Cost(\( \text{Object} \))

\[ \text{Status}[\text{Object}] := \text{seen} \]

For each \( \text{Child} \in \text{Children}[\text{Object}] \)

Case \( \text{Status}[\text{Child}] \) Of

\( \text{unseen} \): Compute-Cost(\( \text{Child} \))

\[ \text{Add-Childs-Cost}(\text{Object},\text{Child}) \]

\( \text{done} \): \( \text{Cost}[\text{Object}] := \text{Cost}[\text{Object}] + \text{Cost}[\text{Child}]/\text{Fanin}[\text{Child}] \)

\( \text{seen} \): \( \text{Rep}[\text{Object}] := \text{Earlier}(\text{Rep}[\text{Object}],\text{Child}) \)

\[ \text{Fanin}[\text{Object}] := \text{Fanin}[\text{Object}]-1 \]

\[ \text{Internals}[\text{Object}] := \text{Internals}[\text{Object}] \cup \text{Child} \]

\( \text{traced} \): \( \text{Rep}[\text{Object}] := \text{Earlier}(\text{Rep}[\text{Object}],\text{Rep}[\text{Child}]) \)

30

End

if Rep[Object] = Object then
    Status[Object] := traced
    Propagate-Cost(Object,Cost[Object],Fanin[Object])
End

End

Add-Childs-Cost(Parent,Child)

If Rep[Child] = Child Then
    Status[Child] := done
else
    Status[Child] := traced
End

End

Propagate-Cost(Object,Cost,Fanin)

If Status[Object] ≠ done Then
    Status[Object] := done
    Fanin[Object] := Fanin
    Cost[Object] := Cost
    For each Child ∈ Internals[Object] do
        Propagate-Cost(Child,Cost,Fanin)
    End
End
End

The accounting process consists of initializing the accounting state and then calling Compute-Cost on each principal root in turn. The algorithm marks each object done once, and follows every reference once. Thus it runs in $\Theta(V + E)$. 

31
3.4 Evaluation

So far we have defined the notion of sponsorship in terms of strong and weak references, and discussed a number of accounting policies, along with algorithms that can implement them. These algorithms and policies present system administrators with a wide array of possible accounting systems. Let us take a comparative look at each of these systems.

<table>
<thead>
<tr>
<th>Policy &amp; Algorithm</th>
<th>Space</th>
<th>Time</th>
<th>System Overhead</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-pay-fully</td>
<td>(\Theta(1))</td>
<td>(\Theta(PV + PE))</td>
<td>None</td>
<td>Full search each accounting pass. Completely fair and stable, but not accurate unless scaling is used. Scaling results in less stability and can encourage antisocial behavior.</td>
</tr>
<tr>
<td>Exhaustive</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Division-of-cost</td>
<td>(\Theta(V))</td>
<td>(\Theta(PV + PE))</td>
<td>None</td>
<td>Fair and accurate, but not stable. Sharing is encouraged.</td>
</tr>
<tr>
<td>Exhaustive</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All-pay-fully</td>
<td>(\Theta(1))</td>
<td>(\Theta(V + E))</td>
<td>None</td>
<td>Same as exhaustive search, but one principal at a time. Can be optimized for disjoint principals.</td>
</tr>
<tr>
<td>Round-robin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Division-of-cost</td>
<td>(\Theta(V))</td>
<td>(\Theta(V + E))</td>
<td>(\Theta(V)) space</td>
<td>Same as exhaustive search, but one principal at a time. Must keep a table mapping objects to number of sponsors.</td>
</tr>
<tr>
<td>Round-robin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-weight</td>
<td>(\Theta(V + E))</td>
<td>(\Theta(V + E))</td>
<td>Reference counting</td>
<td>Accurate, but not fair or stable. Maintains a reference count for each object.</td>
</tr>
<tr>
<td>Single-pass</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-weight</td>
<td>(\Theta(V + E))</td>
<td>(\Theta(V + E))</td>
<td>None</td>
<td>Single pass algorithm with an extra pre-pass to compute reference counts</td>
</tr>
<tr>
<td>Two-pass</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Policy**: The name of the sharing policy used by this system.

**Algorithm**: The name of the algorithm that implements the sharing policy.

**Space**: The amount of space needed for garbage collection with accounting.

**Time**: The amount of time needed for garbage collection with accounting.

**System Overhead**: Degradation in system performance (time or space) incurred by the accounting system, other than during garbage collection with accounting.
Notes: Descriptions and notes about the accounting system.

Of these systems, two stand out as being the most feasible: the round-robin, all-pay-fully with scaling system, and the two-pass object weight analysis system. Both of these systems use algorithms whose performance is asymptotically equivalent to a copy-compaction garbage collection. Each of these two systems has advantages over the other one. The two-pass algorithm is capable of performing a complete accounting pass with each garbage collection cycle, but the accounting information it generates is neither fair nor stable. On the other hand, the scaled all-pay-fully system is completely fair, and approaches stability as the number of principals grows. However, it takes several garbage collection cycles for such a system to compute complete accounting information. Since the maximum accounting frequency is proportional to the number of principals, this may become impractical.
Chapter 4

Conclusion

This work has explored the problem of determining the degree to which each principal is to be held accountable for the space usage in a persistent object store. Chapter 2 discussed mechanisms for describing the set of objects sponsored by a given principal. Among the mechanisms discussed, we concluded that sponsorship links are the most appropriate, and described a number of system features necessary for their implementation. Chapter 3 discussed the process of accounting itself, comparing a number of different accounting policies in terms of their fairness, accuracy, and stability. We discussed the algorithms that implement these policies, and present the single-pass algorithm for object-weight analysis. Lastly, we examined the set of possible accounting systems (policies paired with algorithms) and selected two that might be appropriate for actual use in a persistent object store.

4.1 Future Directions

The concept of sponsorship links, and the systems that employ them, can be expanded in a number of directions.

4.1.1 Programming Language Support

The actual incorporation of sponsorship links into programming and query languages has yet to be implemented. A simple compiler might require the implementor of an abstract data type to declare explicitly every instance variable as strong or weak. More complex systems might employ reference-strength inference rules to optimize an implementation by making unnecessarily strong variables weak, or find implementation errors that cause weak links to dangle trivially. The notion of reference strength might permeate the type system of a high-level language: language constructs could exist to allow programmers to require that clients of a particular procedure or method hold
some or all of its arguments in strong references. A fundamental type library might have “strong” and “weak” implementations of the same type, such as “strong arrays” that sponsor all of their elements and “weak arrays” that do not. A language that supports parameterized types (e.g., CLU and C++), might allow the strength of certain instance variables to be a parameter of the type. These are only some of the considerations to be made by the language designer when including reference strength in a programming language.

4.1.2 Fine-Grained Sponsorship

As described in [Drexler 88], it may be possible to extend the notion of reference strength to include multiple levels, rather than simply two. A particular instance variable might be strong as long as the cost of sponsoring the referent does not exceed a particular maximum. When that maximum is exceeded, the instance variable weakens, allowing the referent to be reclaimed. Using the object-weight accounting algorithm, this extension might be possible; a reference would “break” if it was subjected to too much load. In such a system, it is possible for the data graph to enter states in which more than one reference is “overloaded,” but the breaking of any one “overloaded” reference will cause sufficient deletion of objects such that the other references are no longer overloaded. In these situations, the policy for determining which of these references to “break” must be decided. With some work, an extension of the single-pass or two-pass algorithm might be developed to make this feasible.

4.1.3 Parallelism

The brute-force search used for full-cost accounting can be parallelized easily. Each thread could account for a different principal. Likewise, a round-robin search could parallelize to account for several principals at a time, greatly increasing the frequency of accounting. In a system with many threads, a single principal’s search could be divided amongst several threads. In the latter case, some care must be taken to assure that two threads do not each visit and count the same object, causing that object to be added in twice. To avoid this, threads must share the same accounting information, making notes for each other when an object has already been visited. When the number of processors is proportional to the number of principals, parallel computation might make the exhaustive-search and round-robin algorithms more computationally palatable.

Because of its depth-first nature, the single-pass algorithm does not immediately lend itself to parallel implementations. The algorithm needs to know the order in which objects are visited by the depth-first search. If the search were to be conducted by multiple parallel threads, the problem arises when a thread finds an object that has already been visited by a different thread. In this case, some serialization mechanism must be employed to determine whether a serial implementation would have found a reference between two seen nodes. The serialization mechanism must establish a
total order for all threads. For any two threads $A$ and $B$, it is the case that a serial implementation will follow the entire path of execution of $A$, to its completion, before beginning to follow the path of execution of $B$, or vice versa. Thus, when the search conducted by one thread “collides with” the search conducted by another, it must first be determined which thread is first in the serial order. If the thread which reached the object first is also the “later” thread in the serialization, some backtracking must be performed to determine whether a cycle has been found. As a result, this algorithm could prove less efficient than the serial version.

4.1.4 Distribution

In any distributed object-oriented database, it is necessary for each node to know which local objects are referenced externally. Otherwise, a local node would never be certain whether an object was truly garbage and capable of being reclaimed. With this knowledge, however, a node can garbage collect its local objects, treating external references as sponsorship roots. When an external reference is deleted, the node of the referent should be informed, so that it can reclaim any objects that are not otherwise sponsored. The problem of detecting and reclaiming distributed cycles remains, and has not been satisfactorily solved to date.

In a data-graph-implicit sponsorship environment, a node might occasionally (perhaps only when requested) determine the total cost of externally-referenced objects (including their sponsored subordinates) and report this cost to the nodes where the external references originate. This allows a node to determine the cost of non-local objects when accounting for a particular principal. However, a distributed cycle could cause the process of accounting to be thrown into an infinite loop. To avoid this, some protocol must be established for an accounting search “thread” to cross multiple nodes, marking the objects it visits as it goes. More than one of these conceptual threads could scan the data graph in parallel, although this approach raises its own concerns, which are discussed in 4.1.3 above.

---

1 In the case where a thread splits in two, the thread and its two children are all considered different threads for the purpose of this assertion.

2 see [Shapiro 90]
Bibliography


