# Storage Reclamation in Object Oriented Database Systems

*Margaret H Butler*

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California 94720

*ABSTRACT*

When providing data management for nontraditional data, database systems encounter storage reclamation problems similar to those encountered by virtual memory managers The paging behavior of existing automatic storage reclamation schemes as applied to objects stored in a database management system is one indicator of the performance cost of various features of storage reclamation algorithms The results of modeling the paging behavior suggest that Mark and Sweep causes many more input/output operations than Copy-Compact A contributing factor to the expense of Mark and Sweep is that it does not recluster memory as does Copy-Compact If memory is not reclustered, the average cost of accessing data can go up tremendously Other algorithms that do not recluster memory also suffer performance problems, namely all reference counting schemes The main advantage of a reference count scheme is that it does not force a running program to pause for a long period of time while reclamation takes place, it amortizes the cost of reclamation across all accesses The reclustering of Copy-Compact and the cost amortization of Reference Count are combined to great advantage in Baker's algorithm This algorithm proves to be the least prohibitive for operating on disk-based data

## 1. Introduction

Although relational data base systems appear to be very successful in business data processing environments, more general data base capabilities are needed in engineering applications, CAD/CAM systems, and expert data base environments The exact form of these capabilities and what sort of a software system should provide them is the subject of considerable controversy Specifically, Object-Oriented Database Systems (OODBS) [Zani86, Cope84, Derr85], persistent programming languages [Cock84, Butl86a, Mish84], extensible data base systems [Ston86], and persistent virtual memory [That85] have all been proposed to support these nontraditional database applications The data in these applications can be modeled as a collection of objects[1] which are connected into tree or graph structures as supported by object-oriented programming languages such as Smalltalk [Gold83] and CommonLOOPS [Bobr86] Hence, the proposals to support nontraditional data borrow heavily from object-oriented programming languages

In this application area, objects can be inserted or deleted rendering other objects inaccessible This situation may lead to data being in the database that is not accessible by the application All objects that may contain references to other objects display this characteristic Therefore, all systems providing data management for these objects potentially have this problem Figure 1 depicts a sample scenario that could arise in such systems A set of interconnected objects (represented by shapes) are stored in a database on the disk The *CIRCLE* and the *FOOTBALL* represent the roots[2] of the data structure If the User Program deletes the *CIRCLE*, then the *CIRCLE* is no longer accessible and can be deleted from the database The *STAR* which is contained in the *CIRCLE* is still part of the picture because it is in the *FOOTBALL* but the *FIGURE-8* is not part of the picture any more Ascertaining that the *FIGURE-8* should be deleted but the *STAR* should not is the process of storage reclamation or garbage collection

Storage can be reclaimed either by an automatic method or by issuing explicit delete instructions Programming environments that encourage complex structures provide automatic storage reclamation (e g CommonLOOPS and Smalltalk) On the other hand, current data managers delete items only when instructed to do so However, as the structures in a database grow larger and more complex, determining what objects are no longer reachable becomes exceedingly difficult to do manually Hence, garbage collection should be an integral part of

This research was sponsored by the U S Air Force Office of Scientific Research under Grant 83-0254, the National Science Foundation under Grant 85-04633, and a fellowship from TRW

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise, or to republish, requires a fee and/or specfic permission

© 1987 ACM 0-89791-236-5/87/0005/0410 75¢

[1] An *object*, for the purposes of this paper, is a structure that contains pointers or other nonsimple data types (integers and strings are examples of simple data types)

[2] A *root* is an object that is reachable absolutely not only via other objects

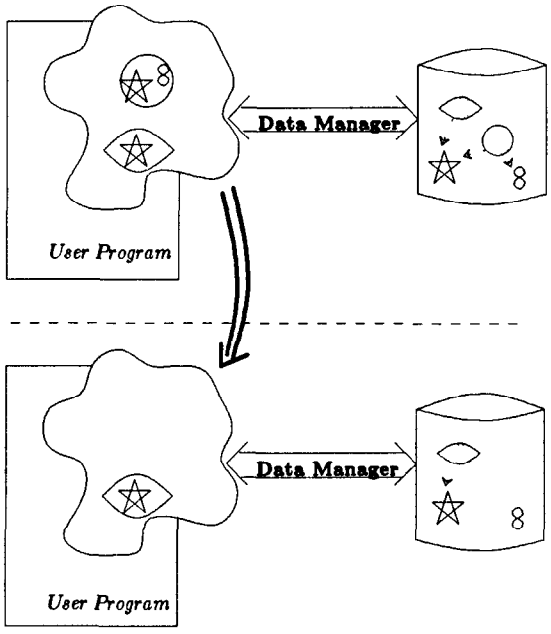


**Figure 1: Sample Object Scenario**

future object-oriented data managers and persistent systems In fact, PO [Mish84], GemStone [Cope84], and POMS [Cock84] already incorporate automatic storage reclamation schemes into their systems

The purpose of this paper is to study the performance of storage reclamation algorithms in an object-oriented data base environment Since persistent programming languages, persistent virtual memory, and extensible database systems have many of the same characteristics as an OODB, the results of this study will also apply to such systems

Storage reclamation is a widely studied problem, therefore many algorithms appear in the literature All algorithms were designed for main memory or virtual memory environments where many data objects have a very short lifetime This is not the case in an OODB environment By analyzing storage reclamation algorithms as they apply to data stored in a database system, features of the algorithms can be identified that have prohibitive performance costs or that are beneficial to performance These results can help guide the design of storage reclaimers for database systems

The remainder of this paper is organized as follows Section 2 contains the model used for algorithm evaluation, descriptions of the algorithms considered and the results of the analysis comprise Section 3, lastly, Section 4 contains the conclusions that can be drawn from the study

## 2. The Model for I/O Cost Evaluation

The data structures used in this study are LISP *cons cells* Each cons cell contains two pointers (like a binary tree) A pointer can refer to a leaf or to another cell Each cell may be pointed to by an unlimited number of cells Cons cells combine to form *lists* Cons cells are good objects to study because they have a uniform structure, and can be combined into complex, interrelated structures The results reported herein scale to larger more complex objects, although the proof of this is beyond the scope of this paper

The LISP lists used in this study are assumed to be stored in a relational DBMS as implemented in Polymnia [Butl86a], a persistent LISP prototype A simplified version of the database schema is shown in Figure 2 The ObjectRelation stores list cells individually, each representing one object Since the right and left pointers can refer to different types, a tag field (*type*) is used to indicate the sort of object pointed to If *type* = "leaf", the pointer refers to a leaf If *type* = "pointer", the *value* field indicates the id of the cell in the ObjectRelation that is pointed to The RootRelation contains the persistent LISP symbols or atoms from a LISP application A tuple in the ObjectRelation is *reachable* only if it is referenced by a root or by an object reachable from a root

The algorithms are compared on their expected amount of I/O Each algorithm has been coded in a query language that resembles Quel* [Ston83] The paging behavior of each implementation is modeled by an equation which calculates the expected number of pages that will be read and written during the running of the corresponding reclamation scheme In order to capture the I/O cost with a single number, each page read is assumed to have unit cost and each page write, a cost of 2 This is in accordance with folklore that disk writes are twice as expensive as disk reads [3] Each equation includes

[3] The actual cost of a write for these particular algorithms may be more or less than twice the cost of a read This assumption affects the totals reported here but does not affect the relative performance of the algorithms since the number of objects read by any particular algorithm is approximately equal to the number written

| ObjectRelation | | | | |
|---|---|---|---|---|
| id | *left* | | *right* | |
| | ltype | lvalue | rtype | rvalue |

| RootRelation | | | |
|---|---|---|---|
| id | name | type | value |

**Figure 2: Database Schema for Lists**

the probability that a page has been previously read and given that it has been read, the probability that it is still buffered in main memory Therefore, the equations express the expected amount of physical I/O

The modeled parameters and the associated value set for each is shown in Table 1 and described below

- The number of objects, **N**, represents the number of tuples in the ObjectRelation This indicates the size of the data structure The number of objects in this study ranged from 5500 (a small database) to 1 million (a medium large database) A structure of 1 million objects is large enough to cause some of the algorithms to exhibit unreasonably high I/O cost
- The number of roots, **R**, specifies the number of tuples in the RootRelation In order to isolate how the data structure parameters effected performance, the number of roots was set to 1
- The number of objects per page, **NP**, represents the number of tuples in the ObjectRelation that fit on a single disk page The tuples in the ObjectRelation for Polymnia utilize approximately 50 bytes so 20 tuples fit on a 1024 byte page The objects per page is set to 20 for this study
- The number of roots per page, **RP**, represents the number of tuples in the RootRelation that fit on a single disk page The tuples in the RootRelation for Polymnia utilize 28 bytes so approximately 35 tuples fit on a 1024 byte page RP equals 35 in this study
- The number of buffer pages, **BP**, signifies the number of main memory pages that the DBMS uses for buffering disk pages Although the number of buffer pages was varied from 3 to 15, it made less than 2% difference in the I/O cost for all the cases tested In a few isolated cases, BP=50 was tested and resulted in a 10% improvement An additional study has been completed in which a larger number of buffer pages has been studied, but the results of this study are beyond the scope of this paper For the results reported here, the number of buffer pages has been left out of the graphs
- The number of accesses, **A**, expresses the number of times that any object in the database is read, written, created, or destroyed It's value was computed relative to the number of objects in the database and ranged from **N** up to 30 times **N**
- The update ratio, **UR**, represents the probability that an access alters an object, that is, the expected number of updates divided by the number of accesses The update ratio ranges from 1/10 down to 0 An UR of 0 indicates a read-only database
- The **UDRatio** expresses the expected number of objects that become unreachable after an update, that is, the expected number of unreachable objects divided by the number of accesses The UDRatio ranges from 1/100 (only 1 in every 100 updates causes an object to become unreachable) to 1 (each update causes 1 object to become unreachable)
- The delete ratio, **DR**, is the expected number of objects that will be deleted per access It is calculated by multiplying the UDRatio by the update ratio

| Parameter | Values | Notes |
|---|---|---|
| N | {5500 12,500 25,000 100,000 500,000 1,000,000} | Number of objects |
| R | 1 | Number of objects in the root set |
| NP | 20 | Number of objects per page |
| RP | 50 | Number of roots per page |
| BP | {3 7 15} | Number of buffer pages |
| A | {1 3 10 30} * N | Number of Accesses (includes reads, updates, & deletes) |
| UR | { 1 01 001 0} | Update Ratio |
| UDRatio | {1 4 100} | Update to Delete Ratio |
| DR | UR/UDRatio { 1 00001 0} | Delete Ratio |
| IR | DR | Insert Ratio |
| Shape | {full bushy sparse} | |
| Density | {1 1 3 2} | |

**Table 1: Parameters modeled and their values**

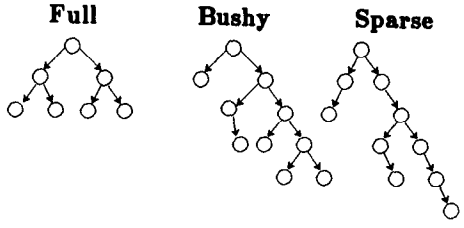


**Figure 3: Examples of lists with varying fullness values**

- The insert ratio, **IR**, is the expected number of new objects divided by the number of accesses For this study, the database is assumed to be in a steady state, that is, the expected number of new objects equals the expected number of deleted objects The IR value is calculated as UDRatio*UR, the same as the delete ratio
- The form of the data structure is determined by the value of **Shape** The possible values are *full*, *bushy*, and *sparse* If the structure has a full shape then the path from the root to any leaf equals log N ±1, and, the number of *right values* that contain nonleaf pointers equals the number of *left values* that contain nonleaf pointers If the structure is bushy, the number of *left values* that contain nonleaf pointers is 7/8 of the number of *right values* that contain nonleaf pointers, and, the path from the root to the rightmost leaf (*right height*) equals N/16 and the path from the root to the leftmost leaf (*left height*) equals N/32 A sparse structure has a left height of 5 and a right height of N/10, and, the number of *right values* that contain nonleaf pointers is 2 times the number of *left values* that contain nonleaf pointers Figure 3 depicts a small example of each shape The examples only indicate the general form of the lists since the number of cells in each example is too small to exactly fit the rules used to define the shapes
- The **density** signifies the number of right or left values that contain nonleaf pointers divided by the number of objects The minimum density is 1 since each cell must be referred to by at least 1 other cell in order to be reachable Since each object has at most 2 pointers, the maximum density for this study is 2 Figure 4 depicts some sample structures that have the densities used in this study

Another factor that can affect the I/O performance of the algorithms is the organization of the objects on pages Placing objects that will be accessed together on the same page will reduce the number of pages that must be fetched from disk when accessing the objects That is, the cost of accessing NP objects that are perfectly clustered on a single page is 1, whereas the cost of accessing NP objects that are scattered across pages is NP As clustered objects are deleted from pages, internal fragmentation of the cluster will occur This cluster fragmentation causes the average cost of accessing objects to increase For example, if NP/2 objects are deleted from each page on which NP objects were clustered, two pages must be read in order to access NP objects

Figure 5 depicts how cluster fragmentation degrades the cost of accessing objects over time The three different lines in Figure 5 represent the expected average access cost for delete ratios of 1/10, 1/100, and 1/1000 (from left to right) The expected average cost of accessing an object when NP objects are clustered on each page is 1/NP, which indicates that only 1 page is read to access NP objects That is, the minimum cost to access one object is 0 05 which is 1/NP when NP equals 20, as is the case in this study As objects are deleted, fewer clustered objects can be accessed from a single page read, therefore the expected cost of accessing objects approaches 1 A cost of 1 indicates so many objects have been deleted that no objects are clustered on pages, therefore each object access causes 1 page read The greater the number of objects deleted (i e the higher the delete ratio), the quicker the expected cost of accessing an object reaches 1

A method of clustering that is efficient for LISP programs is to place objects together that are linked via right pointers (these are called CDRs in LISP) LISP programs tend to access data in a depth-first fashion along CDRs [Bobr79] Organizing cells onto a single page that are linked together via CDRs is called CDR-coding This can be thought of as clustering cells together according to the depth-first traversal of the structure Objects can be clustered according to the breadth-first traversal, too Both breadth-first and depth-first clustering are explored in this study

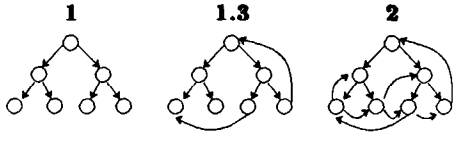


**Figure 4: Examples of lists with varying density values**

In performing the analysis, only a few parameters are varied at a time Unless otherwise specified, the settings are

UR = 0 1
ProbDelete = 0 025
N = 1,000,000
Shape = bushy
Density = 1 3
A = 1,000,000

These numbers represent a middle ground a medium sized database, some updates and deletes, a few doubly connected objects, and one access per object on average In addition, the objects are assumed to be clustered in a depth-first fashion when the analysis begins since current LISP systems tend to allocate cells depth-first

### 3. Algorithms Explored

Many algorithms have been developed to perform garbage collection [Cohe81] Environments that have automatic storage reclaimers operate in two modes one in which garbage is being *generated* and one in which the memory is being *scavenged* to reclaim the storage space used by the garbage On a single processor, during the generation phase a program is running, during the scavenge phase the program must pause Incremental algorithms break up scavenging into small pieces so the pauses are shorter but more numerous All objects not reachable from the root set of objects should be reclaimed during garbage collection The following are critical cost points that should be minimized and, therefore, give a basis for algorithm comparison

the length of pauses
the expected average cost to access objects, and
the total garbage collection cost

The average access cost depends upon whether or not the algorithm supports clustering

#### 3.1. Mark and Sweep

In Mark and Sweep algorithms, scavenging has two phases a mark phase and a sweep phase In the first phase, the data is traversed marking everything that is accessible from the root set In the second phase, a pass is made over the memory deleting everything that is not marked Using Mark and Sweep, a running program can experience long pauses during the scavenging of memory In addition, this algorithm results in much cluster fragmentation since the deleted elements will be interspersed with nondeleted elements

The first phase of Mark and Sweep is inherently recursive A few methods have been explored to implement recursion in a DBMS [Banc86b], two of which are explored here repeated execution [Ston83], and transitive iteration [Banc86a] Figure 6 depicts these two different methods in pseudo-Quel The first method executes the Quel operator **replace** repeatedly until the query results in no change to the database The second method implements transitive iteration in Quel by keeping a temporary relation, the *MarkRelation*, that holds the values that were updated at each iteration and only uses those tuples in the next iteration

The expected I/O cost of three different methods of processing Mark and Sweep have been explored **Mark&SweepA** represents the costs of using the repeated execution as depicted by Method 1 **Mark&SweepB** does the transitive closure as expressed by Method 2 **Mark&SweepC** presumes the transitive closure method of Method 2, with the assumption that the temporary relation (MarkRelation) is kept in main memory, hence, no

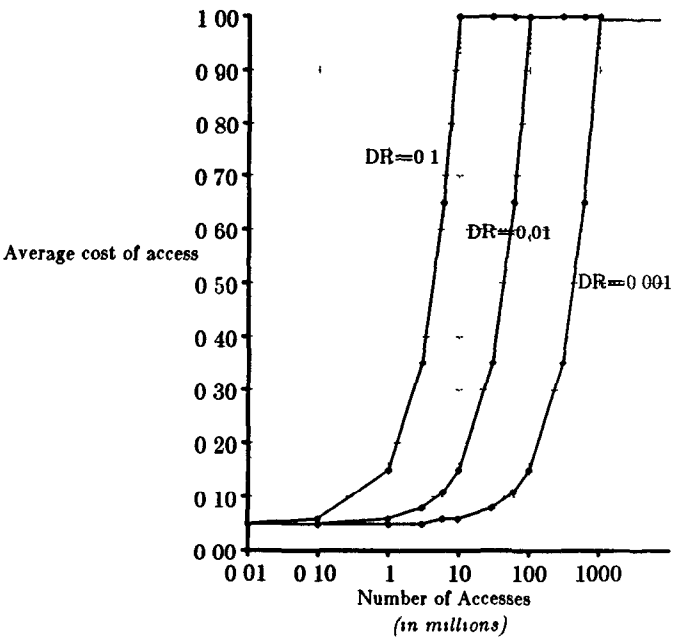


**Figure 5: Access cost as objects become unclustered**

```
/* Method 1 */
range of o,o1 is ObjectRelation
range of r is RootRelation

/* Step 1  mark all objects reachable from the root */
replace (o.mark = TRUE)
where r.type = "pointer" and r.value = o.id

/* Step 2  mark all objects pointed to by a right or left pointer of */
/*              an already marked object, until all are marked */
repeat
        replace (o.mark = TRUE)
        where (o1.right_type = "pointer" and o1.right_value = o.id) or
              (o1.left_type = "pointer" and o1.left_value = o.id)
until no changes to o

/* Step 3  Delete all unmarked (hence unreachable) objects */
delete o where o.mark = FALSE

/* Step 4  Unmark all retained objects */
replace (o.mark = False)

-----------------------------------------------------------

/* Method 2 */
range of m,m1 is MarkRelation

/* Mark all objects pointed to by the root as reachable by retrieving them into MarkRelation */
append to MarkRelation (id = r.value, mark = TRUE)
where r.type = "pointer"

/* While there are objects whose pointers have not been followed */
while count(where m.mark=TRUE) != 0 do {
        /* "Mark" all objects reachable from objects marked TRUE and mark them FALSE */
        /* Mark them only if they haven't been marked before */
        /* FALSE indicates the object has not yet had it's pointers followed */
        append to MarkRelation (id = o1.id, mark = FALSE)
        where (o.id = m.id) and (m.mark=TRUE) and
                [ (o.left_type="pointer" and o.left_value=o1.id) or
                (o.right_type="pointer" and o.right_value=o1.id) ] and
                o1.id != ANY(m1.id)

        /* All objects whose pointers were just expanded from need not be considered again */
        /* All objects not yet expanded from (FALSE) are to be expanded from next (TRUE) */
        replace (m.mark=SAVED) where m.mark=TRUE
        replace (m.mark=TRUE) where m.mark=FALSE
}
/* Delete all objects that are not represented in MarkRelation */
delete o where o.id != ANY(m.id)
```

**Figure 6: Implementations of Mark and Sweep**

I/O cost is incurred from accessing it The equation that models the expected I/O cost of Mark&SweepA is described in detail in Appendix A All of the cost equations are fairly similar so only Mark&SweepA is specified

Figure 7 shows the cost of using each Mark and Sweep implementation as the number of cells increases for full, bushy, and sparse lists For Mark&SweepB and Mark&SweepC, the cost of reclamation only varied 2% over the different values for Shape so the graphs have been combined Although the cost of Mark&SweepB and Mark&SweepC appear to be similar, this is only an artifact of using a logarithmic scale Mark&SweepB is twice the cost of Mark&SweepC because the cost of accessing the MarkRelation is zero for Mark&SweepC Mark&SweepA, for sparse and bushy structures of 1 million cells, has an expected I/O cost of 23 billion A typical, random access, high speed disk can process about 50 page requests per second, hence Mark&SweepA, in this case, would take approximately 15 years to do I/O In the case of full structures, it would take slightly more than 2 days of I/O In contrast, Mark&SweepC would take about 10 hours of I/O for one million objects

The cost formulas shown in Figure 7 assume that the cells are depth-first clustered before the reclamation scheme executes Since Mark and Sweep does not copy the list to retain clustering, over time the cells will become unclustered Figure 8 shows the readjusted performance of Mark and Sweep when no clustering exists The cost increases by 50% for Mark&SweepA on full structures, Mark&SweepB on all structures, and Mark&SweepC on all

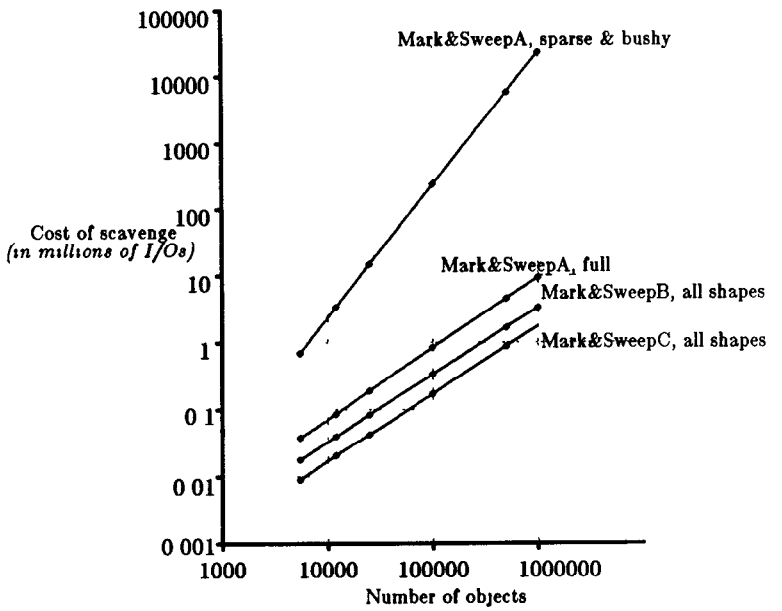


**Figure 7: Cost of Mark and Sweep**

structures Mark&SweepA on bushy structures doubles in cost and on sparse structures triples in cost Therefore, Mark&SweepA on sparse structures requires approximately 45 years of disk I/O when one million unclustered objects are in the database

### 3.2. Copy-Compact

Using a copy-compact algorithm [Minsky63], the root set of objects is copied into an unused portion of memory All objects that are reachable from the root set are also copied into this unused portion of memory The entire memory in which the data used to reside is then reclaimed Copy-Compact schemes require twice as much memory as is occupied by data the memory where the data currently resides (*oldspace*) and a portion equal in size to be used for copying (*copyspace*) After each iteration of the algorithm, copyspace becomes oldspace and oldspace becomes copyspace

As with Mark and Sweep, long pauses occur when the scavenging phase takes place because the copying takes time proportional to the number of objects plus the number of references between objects No cluster fragmentation results from using this method because objects are reclustered in memory as they are copied The memory will be efficiently clustered if the objects are copied in the same order as accesses occur Copying in this manner clusters objects together on a page that will be accessed together

A Copy-Compact algorithm that creates a depth-first cluster and one that creates a breadth-first cluster are examined using the implementation techniques described for Mark&SweepB and Mark&SweepC a disk-based temporary relation is used to implement a transitive closure for *BreadthFirstB* and *DepthFirstB*, and, a physical main memory relation is used for *BreadthFirstC* and *DepthFirstC* The method of query evaluation assumed for Mark&SweepA (i e repeated execution of the replace until no tuples are altered) is not explored due to its very poor performance

Figure 9 graphs, for the three Shape values, the cost for the Copy-Compact implementations as the number of objects increases DepthFirstB is more costly than BreadthFirstB because depth-first traversal requires two temporary relations one to handle the transitive closure involved in stepping down the right pointers, and the other to keep track of the left pointers that need to be followed Breadth-first traversal treats right and left pointers the same, therefore, one temporary relation is sufficient

Since DepthFirstB is dependent upon both the left and the right height, and the average height[4] of a full structure is so much smaller than that of a sparse or bushy structure (log N versus (N+5)/10 or 3N/32), full structures have lower cost than bushy structures for DepthFirstB DepthFirstB for sparse structures has a low cost because DepthFirstB is designed to traverse right pointers all at once, so if a structure has lots of right pointers, as is the case with sparse structures, and the cells are clustered on their right pointers, which depth-first copying enforces, a large cost is not incurred The most costly of the data structures, the bushy shape for DepthFirstB, would take approximately 3 years of disk I/O BreadthFirstC and DepthFirstC have equivalent performance since DepthFirstC's extra temporary relation is also kept in physical memory, hence does not incur the extra I/Os which make DepthFirstB more costly BreadthFirstC and DepthFirstC would take about 7 1/2 hours of I/O each, in con-

[4] average height = (left height + right height)/2

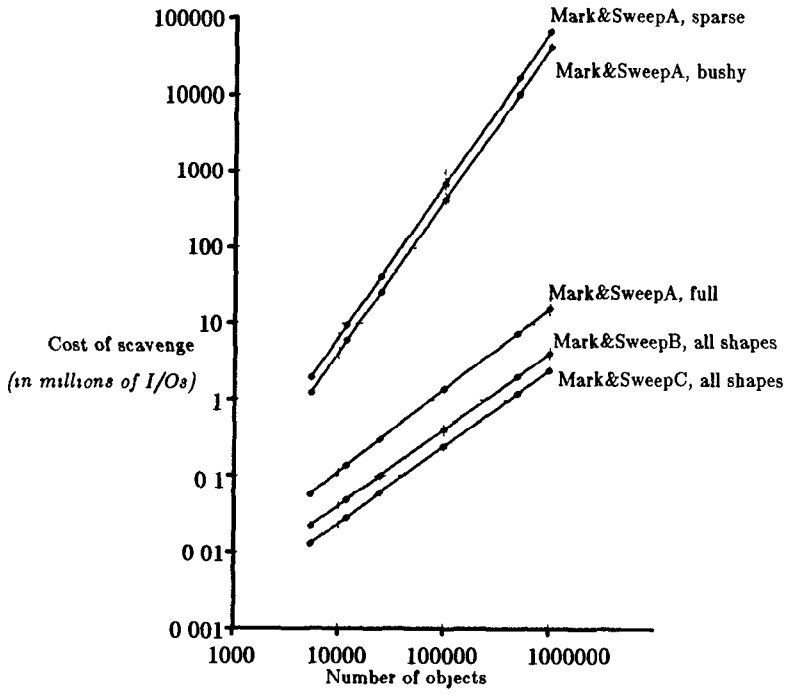


**Figure 8: Unclustered Performance of Mark and Sweep**

trast with unclustered Mark&SweepC which would take 15 hours of I/O

The objects are clustered efficiently for the storage reclamation algorithms if the access pattern of the algorithm matches the object placement pattern This will occur if the object allocation scheme happens to allocate objects on pages that match the reference pattern of the storage reclamation scheme Since LISP systems tend to allocate depth-first, how well BreadthFirst Copy-Compact performs on a depth-first clustered structure is of interest The I/O costs of the BreadthFirstB and BreadthFirstC, if the cells are actually clustered in a depth-first pattern, are indistinguishable from the costs if the cells are clustered in a breadth-first pattern Therefore, the clustering pattern of the data does not have to match the access pattern of the garbage collection algorithm in order to benefit performance

### 3 3. Baker's realtime

Baker [Bake76] developed a version of copy-compact that breaks up the job of object copying into pieces so that the copying can be done gradually In each piece a bounded number of objects are copied, thereby limiting the length of the pauses in a program Each time a new object is allocated, the pointers from k objects in copyspace are traversed [5] If a pointer refers to an object in oldspace, the object is copied into copyspace and the pointer is updated to refer to the object's copyspace location Since pointers may still exist that refer to the object in oldspace (after it has been moved to copyspace), a marker must be left in the oldspace object that indicates the object now resides in copyspace Each time an object in oldspace is accessed, it is also moved into copyspace When all objects in copyspace have been traced, all the reachable objects have been copied into copyspace, and the spaces are exchanged oldspace becomes the new copyspace and the root set of objects is copied into it, and copyspace becomes oldspace

In the database implementation, oldspace and copyspace can be two separate relations Since database objects can be referenced by logical identifiers not addresses, an object can have the same logical identifier in both relations Hence, this implementation allows no markers to be left in oldspace This simplifies the lookup procedure an object should be looked for first in copyspace and only if it is not found there is oldspace checked The lookup cost increases by 50% since on average half of the objects will require a double lookup Other than the increased access cost, Baker has a total cost similar to the breadth-first version of Copy-Compact although, because the copying is broken up into chunks, Baker continuously reclusters the objects on disk

### 3.4. Generation-Scavenge

In order to reduce the number of objects in memory that need to be considered when doing a copy, the Generation-Scavenge schemes assign ages to objects [Unga85] The age groups are determined by the algorithm implementor A separate portion of memory is allocated for objects of each age group Each age group's root set consists of the objects in this age group that are pointed to by objects in older age groups The younger the object, the more likely it is to become garbage

[5] k can be set by the algorithm implementor or by particular users to limit pauses to an imperceptible amount of time

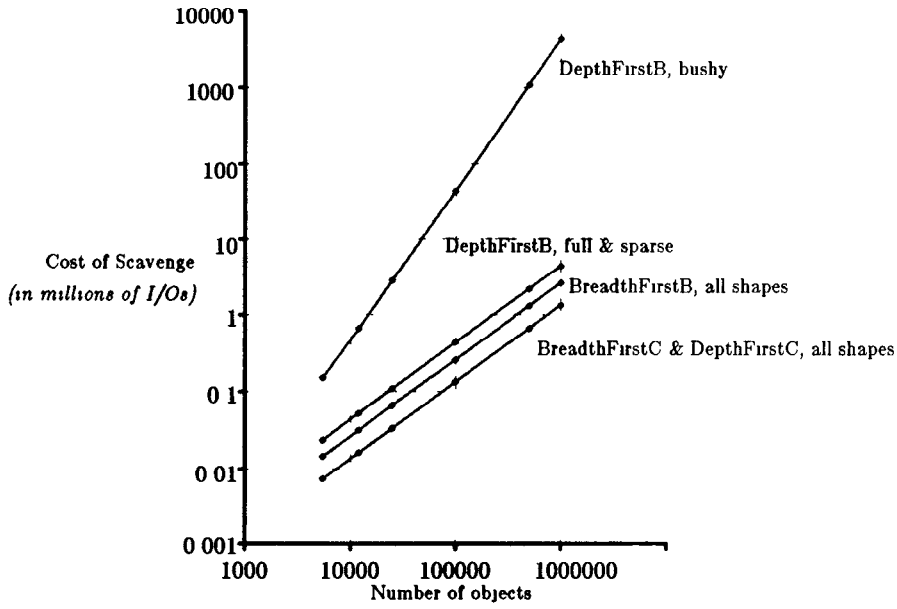


**Figure 9: Cost for Copy-Compact**

[Lieb83] so young spaces are scavenged more often than older spaces That young objects become reclaimable sooner is caused by the use of temporary variables in programming languages Objects that reside in the oldest space are considered to be permanent objects and their space is not examined very often to see if it contains any garbage The pauses caused by a scavenge are greatly reduced in this scheme since only a subset of the objects are traversed at each scavenge

Distinguishing between temporary and permanent data does not gain much in a persistent environment because all of the objects are permanent Temporary variables are not likely to be stored in a database so Lieberman's result does not apply to young database data Therefore, the assumption that allows Generation-Scavenge to have better performance than other algorithms (namely, that objects become garbage based upon their age) does not hold for database objects Hence, Generation-Scavenge is not explored in this study

### 3.5 Reference Count

If a Reference Count scheme is employed, each object in memory contains a count of the number of objects that reference it Each time a reference is removed, the count is decremented If the count becomes zero, the object gets reclaimed and all objects to which it points must have their counts decremented This in turn may cause an object's count to become zero Hence, removal of a single reference could cause multiple deletes This method tends not to have long pauses since the work is distributed across alterations, but due to the cascading effect of a decrement, the amount of time any alteration takes is highly variable

Reference counting requires extra space in memory to store the count for each object This is very expensive for small objects In addition, this scheme does not detect circular garbage In the simplest case, a circle of garbage will occur if *objectA* references *objectB* and *objectB* references *objectA* but no other object references either *objectA* or *objectB* Both *objectA* and *objectB* have reference counts of one, but neither is accessible from a root object

An update to an object may be changing a pointer Hence, an update may cause two objects' reference counts to change the object that used to be pointed to needs to have its reference count decremented and the new object pointed to must have its count incremented So each update causes two extra writes in order to update the counts Likewise, a delete causes the objects that were referred to by the deleted one to have their counts decremented resulting in two writes Assuming no circular garbage exists, the average increase in access cost incurred because of the increased update cost in a Reference Count scheme is graphed in Figure 10 The base cost to access an object is 1/NP (0 05) which increases based upon the number of accesses that are updates or deletes These amounts shown in the figure do not factor in increases due to loss of clustering Each of the curves has a different value for the update ratio, the delete ratio varies within each curve along the X-axis, and, the Y-axis represents the average I/O cost The increase in access cost ranges from 0, when no objects are updated or deleted, to 1 6, when the update ratio and the delete ratio are 0 1

### 3.6. Logged Reference Count

In order to overcome the variable cost associated with decrementing the reference counts, alterations to reference counts can be logged and the log can be processed to update k reference counts at a time This would put a bound on the number of objects that can be changed by one update Each update will cause one write to the log instead of two writes This does not reduce the number of total I/O (in fact it increases it) but it reduces the manda-

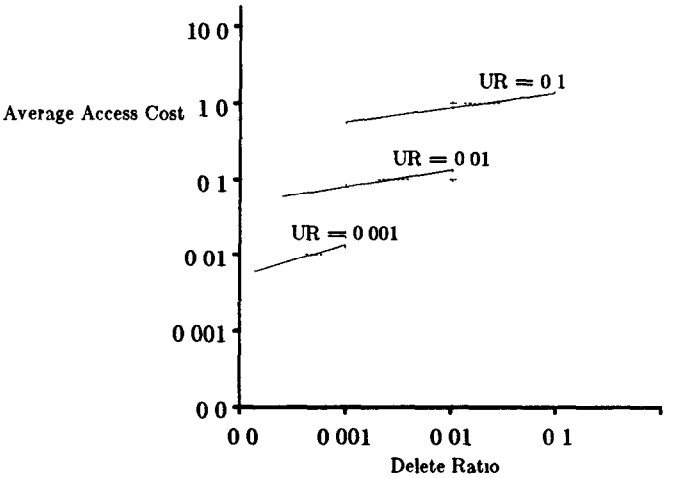


**Figure 10: Increase in Per Access Cost for Reference Count**

tory increase in the average access cost The expected increase in the average access cost when Logged Reference Count is used is shown in Figure 11 These amounts do not include the cost to process the log, only the cost to write to the log Each of the curves has a different value for the update ratio, the delete ratio varies within each curve along the X-axis, and, the Y-axis represents the average I/O cost The cost ranges from 0, when no objects are updated or deleted, to 0 3, when the update ratio and the delete ratio are 0 1 The increase in access cost is independent of DR because each update only causes a write to a log, all updating of reference counts and removal of unreferenced objects occurs at log processing time The cost increase is very low since the reference counts are not actually updated until the log is processed The total cost of this method is shown in the Section 3 8 where it is compared with the other schemes

### 3.7. Deutsch-Bobrow reference count

Deutsch and Bobrow [Deut79] made the observation that most objects have a reference count of one This means that most of the space typically allocated to hold the counts is wasted They proposed keeping a table of the objects that have a zero count (newly created objects) and a table of objects that have a count greater than one The tables hold only the addresses of the objects, no other information When a reference to an object on the multireference table is removed, no information is kept that would indicate whether that was the last reference to the object Therefore, once an object is on the multireference table, it cannot be removed Their scheme operates the same as standard reference counting but uses less space because individual reference counts are not needed, although over time more space may be used because extra garbage may gather due to the permanent effect of being added to the multireference table

Figure 12 graphs the extra cost per access that the Deutsch-Bobrow method incurs Each of the curves has a different value for the update ratio, the delete ratio varies within each curve along the X-axis, and, the Y-axis represents the average I/O cost The cost differs from vanilla Reference Count since the reference counts of objects on the multireference table don't cause writes The cost ranges from 0, when no objects are updated or deleted, to 0 72, when the update ratio and the delete ratio are 0 1 The cost is highly dependent on the number of objects that have multiple references, which for this study was 30% [6]

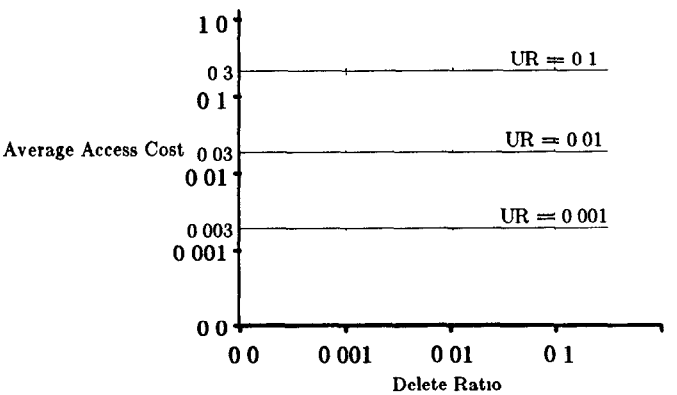


**Figure 11: Increase in Per Access Cost for Logged Reference Count**

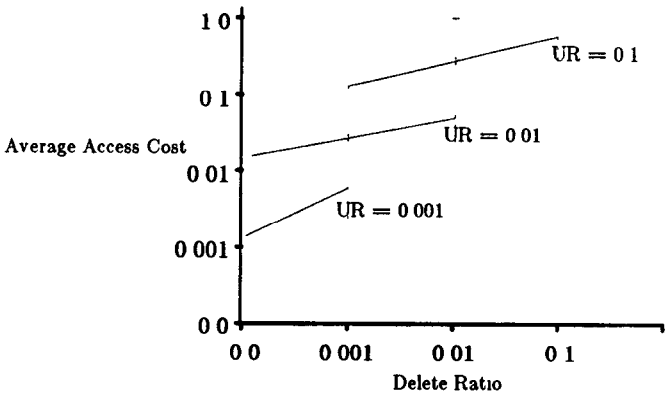


**Figure 12: Increase in Per Access Cost for Deutsch-Bobrow**

### 3 8. Comparing the schemes

As was stated in this section's introduction, the items to consider when comparing the I/O cost of garbage collection schemes are the length of the pauses, the clustering allowed, and the total cost of garbage collection The latter two items can be captured by considering the total cost of accesses plus the garbage collection cost Whether data is clustered or not determines the expected average access cost, adding to that the cost of garbage collection will indicate the entire cost of a particular scheme Dividing this total cost by the number of accesses will give the expected average cost of accessing an object when garbage collection cost is considered an access cost, henceforth, *average joint cost* refers to this conglomerated average cost The average joint cost for Mark&SweepC, DepthFirstC, Baker, Reference Count, LogRefCount, and Deutsch-Bobrow[7] is 1 3, 0 65, 0 53, 1 7, 1 8, and 1 4, respectively, assuming 30 million accesses,1 million objects, an update ratio of 0 1, a delete ratio of 0 025, and assuming Mark&SweepC, DepthFirstC, and Baker complete three scavenges of memory during this interval

The average joint cost of Mark&SweepC, DepthFirstC, and Baker varies depending upon how many complete scavenges occur in the interval Figure 13 shows how the average joint cost varies for Mark&SweepC, DepthFirstC, and Baker as the number of scavenges varies DepthFirstC and Baker have a U-shaped curve because the fewer times that compacting occurs, the higher the average access cost since objects get reclustered less often The bottom of the U-shaped curve indicates when garbage collection should occur in order to minimize the average joint cost for the method This occurs when 5 scavenges are done in DepthFirstC, or, in other words, when 15% of the objects have been deleted, and, when 4 complete scavenges are done for Baker, or 20% of the objects have been deleted In order to keep the average joint cost at a minimum, when DepthFirstC or Baker is employed, they should be executed when 15% or 20% of the objects, respectively, have been deleted Mark&SweepC has a lower joint cost the fewer times it is executed Therefore, the optimal time to run it is dictated by how much disk-space can be allowed to be occupied by garbage Figure 14 depicts the trade-off between the number of disk-pages that can be occupied by garbage and the average joint cost of Mark&SweepC As the number of disk-pages allowed to be wasted on garbage increases, the less often Mark and Sweep needs to run, thereby causing a decrease in the average joint cost

The bar chart in Figure 15 shows the average joint cost for Mark&SweepC, DepthFirstC, Baker, Reference Count, LogRefCount, and Deutsch-Bobrow with varying change and delete rates For comparison purposes, Mark&SweepC and DepthFirstC are activated when 25% of the objects are deleted and Baker does a complete scavenge in the same interval The first bar for each algorithm represents the cost when UR=DR=0 When no changes occur to the objects, no cost for garbage collection is incurred, therefore, this bar indicates just the cost of 30 million accesses, which is the same for all methods (i e 0 05) except Baker As explained in Section 3 3, Baker incurs a 50% increase in access cost, therefore it has an average joint cost of 0 075 The second bar shows the cost when the Update Ratio is 0 01 and the Delete Ratio is 0 0001 Due to the very low update and delete probabilities, the cost is not very high for any of the schemes The Delete Ratio is not high enough to activate Mark&SweepA or DepthFirstC during this interval The third bar represents the cost when the Update Ratio is 0 1 and the Delete Ratio is 0 025 The average joint cost of DepthFirstC and Baker is less than the other schemes due to the decrease in access cost that comes with reclustering the data Baker has the least cost because it is constantly reclustering the data

[6] This number (30%) comes from the density which was set to 1 3 If the average number of references to an object is 1 3, the probability of an object being referenced more than once is 30%

[7] Mark&SweepC is the least costly of the Mark and Sweep implementations, so it is the one used in the comparison DepthFirstC and BreadthFirstC have the lowest cost of the Copy-Compact implementations, since they have equal cost, DepthFirstC is used here for comparison

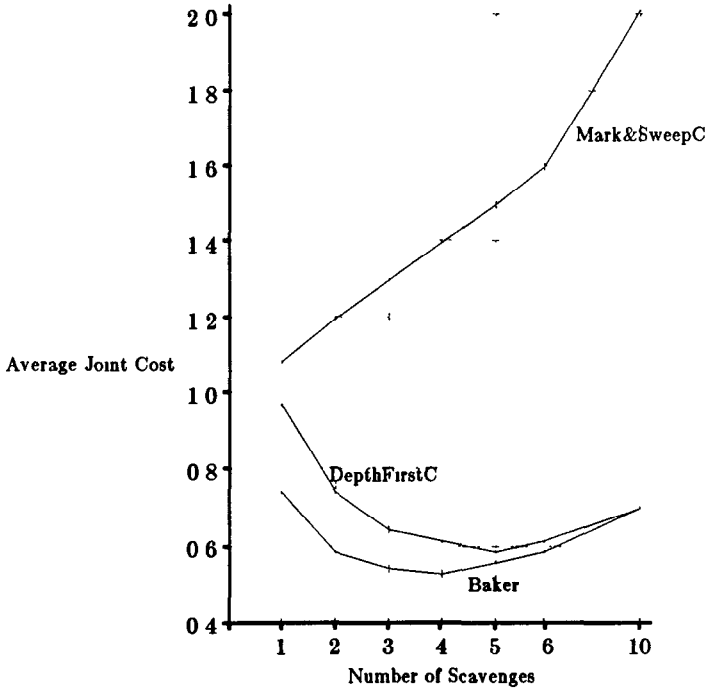


**Figure 13: Average Joint Cost as Number of Scavenges Varies**

#### 4. Conclusions

Many of the criteria for judging an automatic reclamation scheme are the same for virtual memory and disk-based schemes, such as

> Causing long pauses in the program is bad,
> Scavenging incrementally is good,
> Allowing internal fragmentation is bad, and,
> Clustering improves performance

The pauses caused by Reference Count, Logged Reference Count, and Deutsch-Bobrow are much shorter than those caused by Mark and Sweep or Copy-Compact, but the average joint cost of Reference Count is much higher than that of either of the Copy-Compact versions Breaking up Copy-Compact into pieces, as is done by Baker, allows the pauses to be shorter and maintains a low average joint cost

An additional property that should be considered for choosing a storage reclamation scheme for a database is whether the cost of reclamation can be postponed to times when the database is idle Mark and Sweep and Copy-Compact both have this characteristic, Baker and Logged Reference Count allow most of the cost to be postponed If reclamation occurs at idle times, the I/O cost is one indication of how long the idle must last in order to complete the reclamation during this time If the database gets a request for data during reclamation, the reclamation

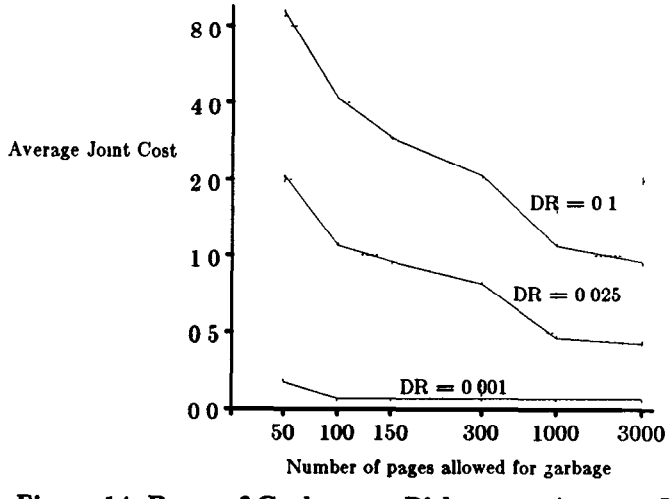


**Figure 14: Pages of Garbage on Disk versus Average Joint Cost for Mark&SweepC**

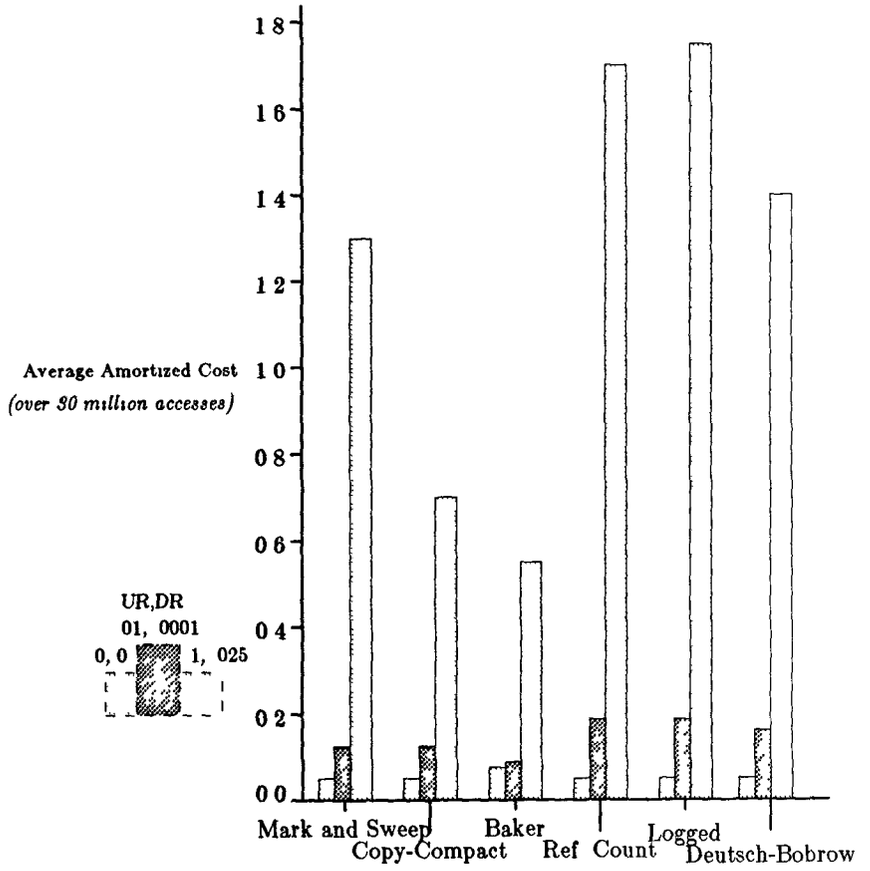


**Figure 15: Cost Comparison with Varying Update and Delete probabilities**

could pause until the database is idle again, or it could force the request to wait The former is preferred so that the pauses realized by a user are minimized, but not all algorithms can be easily interrupted Logged Reference Count and Baker may be interrupted without losing as much work as Mark and Sweep or Copy-Compact since their work can be done incrementally

An issue that needs to be examined is whether locking the entire database is demanded by the garbage collection algorithm The current assumption is that the program cannot access the database during a scavenge This restriction may not be necessary

Weighing the important factors (reclustering, length of pauses, and interruptibility) Baker is the clear winner Even though Baker has the least cost of all the algorithms explored, it still increases the cost to 10 times what it is when no garbage collection is done That is, the minimum average cost to access an object is 0 05 (1/NP), if the Baker method of garbage collection is used the cost increases to 0 53, but, if no reclustering of memory is done, the cost increases to 1 Despite that Baker can cause a reduction in the average access cost because it reclusters, 0 53 page reads per access is high If the chunks of reclamation could be done when the database is not in use or could be done in the background, the average cost to access an object would become 0 075 Users may be willing to pay that much At least the cost of Baker can be used for comparison when designing hybrid schemes to try to reduce the cost further

## *ACKNOWLEDGEMENTS*

Many thanks to my advisor, Michael Stonebraker, for critiquing multiple versions of this paper, the resulting paper is much clearer due to his help In addition, I am grateful to Douglas Terry for helping me with early versions of this paper when understanding it was a real chore Lastly, the reviewers' comments were helpful in pointing out the details that were still unclear Thank you

## *REFERENCES*

Bake76
H Baker, *List processing in real time on a serial computer,* Massachusetts Institue of Technology, Cambridge, Massachusetts (Ocober 1976)

Banc86a
F Bancilhon and S Khoshafian, "A Calculus for complex objects," *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,* pp 53-59 (March 1986)

Banc86b
F Bancilhon and R Ramakrishnan, "An amateur's introduction to recursive query processing strategies," *Proceedings of ACM SIGMOD* , pp 16-52 (May 1986)

Bobr79
D Bobrow and D Clark, "Compact Encodings of List Structure," *ACM Transactions on Programming Languages and Systems* **1**(2) pp 267-286 (October 1979)

Bobr86
D Bobrow, K Kahn, G Kiczales, L Masinter, M Stefik, and F Zdybel, "CommonLoops merging Lisp and object-oriented programming," *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp 17-29 (October 1986)

Butl86a
M Butler, "An Approach to persistent LISP objects," *Proceedings of the Thirty-First IEEE Computer Society International Conference*, pp 324-329 (March 1986)

Cock84
W Cockshot, M Atkinson, K Chisholm, P Bailey, and R Morrison, "Persistent object management system," *Software--Practice and Experience* **14** pp 49-71 (1984)

Cohe81
Cohen, "Garbage collection of linked data structures," *Computing Surveys* **13**(3)(September 1981)

Cope84
G Copeland and D Maier, "Making Smalltalk a database system," *Proceedings of SIGMOD 1984*, pp 316-324 (June 1984)

Derr85
N Derrett, "An Object-oriented approach to data management," *Proceeedings of the Thirty-First IEEE Computer Society International Conference*, pp 330-335 (March 1986)

Deut79
L Deutsch and D Bobrow, "An efficient incremental garbage collector," *Communications of the Association for Computing Machinery* **19**(9) pp 522-526 (September 1979)

Gold83
A Goldberg and D Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley (1983)

Lieb83
H Lieberman and C Hewitt, "A Real-time garbage collector based on the lifetimes of objects," *Communications of the ACM* **26**(6) pp 419-429 ()

Mins63
M Minsky, "A Lisp garbage collector using serial secondary storage," Memo 58, Project MAC, MIT, Cambridge, Masachusetts (December 1963)

Mish84
N Mishkin, "Managing Permanent Objects," PhD Thesis, Department of Computer Science, Yale University, New Haven, Connecticut (1984)

Ston83
M Stonebraker, "Applications of abstract data types and abstract indices to CAD applications," *ACM SIGMOD International Conference on the Management of Data*, (May 1983)

Ston86
M Stonebraker and L Rowe, "The design of Postgres," *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp 340-355 (May 1986)

That85
S Thatte, *Persistent Memory for Symbolic Computers*, Texas Instruments Technical Report 08-85-21, Dallas, Texas (July 22, 1985)

Unga85
D Ungar, "Generation scavenging a nondisruptive high performance storage reclamation algorithm," *Proceedings of Practical Programming Environments Conference*, (April 1984)

Zani86
C Zaniolo, H Ait-Kaci, D Beech, S Cammarata, L Kerschberg, and D Maier, "Object oriented database systems and knowledge systems," in *Expert Database Systems*, ed L Kerschberg,The Benjamin/Cummings Publishing Company, Inc. (1986)

# Appendix A: Explanation of Equation for Mark&SweepA

The equations used to model the I/O cost of the storage reclamation queries include the following assumptions

- The pages that hold the RootRelation are distinct from those that hold the ObjectRelation
- Only whole pages can be read, hence, the number of pages is often rounded up
- Objects are indexed on every field that will be of benefit to the query processing Namely, the *mark* field and the *id* field
- Tuples in the ObjectRelation are assumed to be clustered according to their depth-first traversal as described in Section 2

Table 1A shows variables used in the equations along with how the value is calculated based on the parameters shown in Table 1

The cost of each query in the Mark&SweepA implementation will be described separately for clarity The total cost is the sum of the parts, specifically **Step1 + Step2 + Step3 + Step4**

- The first step in Mark&SweepA is reading the objects in the root set and marking what they point to Recalling that the number of roots in the study is 1 allows the cost of this query to be simplified The root must be read (a cost of 1), and, the object that the root points to must be read (a cost of 1) and marked (a cost of 2 to update the tuple) Hence, **Step1 = 4**
- The replace statement inside the repeat loop is processed repeatedly until no changes are made At each iteration, one more level of objects will be marked A level of objects contains all the objects that occur at the same height in the structure when it is viewed as a tree The replace must be executed once for every level of the structure, plus an additional time when no more cells can be marked This is modeled by

$$\textbf{Step2} = \sum_{i=1}^{i=Height+1} \text{Cost of Replace at level } i$$

The cost of the replace at each iteration includes the cost to read all the cells that are marked up to that point and the cost to follow the pointers that have not yet been followed and mark what each points to The cost of iteration

| Variable | Value | Description |
|---|---|---|
| Height | Log(N) {full}<br>N/16 {bushy}<br>N/10 {sparse} | |
| HeightR | Log(N) {full}<br>N/16 {bushy}<br>N/10 {sparse} | Length of path when following right pointers from the root |
| HeightL | Log(N) {full}<br>N/32 {bushy}<br>5 {sparse} | Length of path when following left pointers from the root |
| Arcs | Density*N | Number of pointers |
| ArcL | Arcs/2 {full}<br>(7/15)*Arcs {bushy}<br>(1/3)*Arcs {sparse} | Number of left pointers |
| ArcR | N/2 {full}<br>(8/15)*Arcs {bushy}<br>(2/3)*Arcs {sparse} | Number of right pointers |
| NR | N/2 {full}<br>(8/15)*N {bushy}<br>(2/3)*N {sparse} | Expected number of objects first reachable via a right pointer |
| NL | N/2 {full}<br>(7/15)*N {bushy}<br>(1/3)*N {sparse} | Number of objects first reachable via a left pointer |
| $X_i$ | X/Height | For any variable X, all X at Height i |

**Table A: Additional variables in equations**

i is modeled by

Cost of Replace at level i =

$\sum_{j=1}^{i} N_j$ * Prob Cell is Not Buffered[8] +

$ArcR_i$ * Prob Cell Right Ptr References is Not Buffered +

$ArcL_i$ * Prob Cell Left Ptr References is Not Buffered +

$N_i$ * Prob Cell Referenced is Not Dirty&Buffered * 2

Since on average the number of objects per level is N/H, the complete summation representing the entire cost of the repeat loop can be simplified, eliminating the double summation This simplification has the following result

**Step2 =**

$\frac{N}{NP} * \frac{H+1}{2}$ /*Clause1*/ * Prob Cell is Not Buffered +

$\frac{H+1}{2}$ /*Clause2*/ *ArcR* Prob Cell Right Ptr References is Not Buffered +

$\frac{H+1}{2}$ /*Clause3*/ *ArcL* Prob Cell Left Ptr References is Not Buffered +

N /*Clause4*/ * Prob Cell Referenced is Not Dirty&Buffered * 2

Clause1 represents the number of pages read from the ObjectRelation Since each marked cell must be read at each iteration and there are (H + 1) iterations, each page of cells is read an average of (H+1)/2 times The probability that the page containing a particular cell is not buffered at this time is represented by

Prob Cell is Not Buffered =

$$\frac{1}{H+1} + \frac{H}{H+1} * MAX\left[0, \frac{\frac{N}{NP} - BP}{\frac{N}{NP}}\right]$$

The first time the query is executed (1/(H+1)) all the pages needed must be read The rest of the times (H/(H+1)), pages will be read only if they are not buffered, as represented by the MAX clause If the number of buffer pages available is greater than the number of pages needed to iterate once (i e if (N/NP - BP) <= 0) the probability that a page is **not** buffered given that it has already been read is 0, otherwise, the probability is the number of ObjectRelation pages not buffered over the total number of ObjectRelation pages Clause2 and Clause3, which involve ArcL and ArcR, include the probability that following a pointer will cause a page read Since the objects are clustered along their right pointers, the probability of causing a read when following a right and left pointer are different The probability that following a right pointer will cause a page read is represented by

Prob Cell Right Ptr References is Not Buffered =

$$1 - \left[\frac{NP-1}{NP} + \frac{1}{NP} * MIN\left[1, \frac{BP}{\frac{N}{NP}}\right]\right]$$

[8] Probability is abbreviated to "Prob" and Pointer is abbreviated to "Ptr" for the rest of the equations

When following a right pointer a page will not need to be read for (NP-1)/NP of the times since cells are clustered via right pointers The first time a cell on a page is referenced (1/NP) it will need to be read only if that page is not buffered from a previous iteration The percentage of pages that are buffered is BP/(N/NP) if BP <= N/NP otherwise it is 1 The probability that following a left pointer will cause a page read is modeled by

Prob Cell Left Ptr References is Not Buffered =

$$1 - MIN\left[1, \frac{BP}{\frac{N}{NP}}\right]$$

Following a left pointer will not cause a page read only if the page the requested cell is on happens to be buffered, which is random since cells are not clustered according to right pointers Clause4 represents the cost of writing the pages that have been marked at each iteration Each cell will get written only once A page will get written if no cell on that page has gotten marked since it was last read from disk This probability is the same as the probability that a page has not yet been read given in the first clause

Prob Cell is Not Dirty&Buffered =
Prob Cell is Not Buffered

- For the delete query, all the pages in the ObjectRelation must be read to see if they contain any unmarked objects because each ObjectRelation page has equal probability of containing objects that are unreachable Hence N/NP pages must be read The expected number of pages that need to be updated because cells on them have been deleted is equal to

  **Step3** =Number of Deleted Cells *
  Prob Cell is Not Dirty&Buffered * 2
  Number of Deleted Cells = N - DR*A

  The probability that the page holding a deleted cell is not dirty and buffered is the same as in the previous step

- In order to change the mark on each object, all the pages must be read and written This is modeled by

  $$\textbf{Step4} = \frac{N}{NP} * 3$$

The total expected I/O cost of of the Mark&SweepA method can be expressed as *Step1 + Step2 + Step3 + Step4* The equations that model the I/O behavior of the other implementations are variations on the above and therefore are not be shown