BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M16

Garbage Collection for Encore/Observer:

by
Katsuji Ishii

Garbage Collection for Encore/Observer:

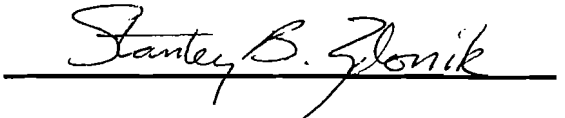
by

Katsuji Ishii
Brown University, 1990

Sc.M.Project

Submitted in partial fulfillment of the requirement for the Degree of Master of Science in the Department of Computer Science at Brown University

December, 1990



Dr. Stanley. B. Zdonik
Advisor

BROWN UNIVERSITY

Department of Computer Science

Master's Project

Garbage Collection for Encore/Observer:

by

Katsuji Ishii

Abstract

In this thesis we will describe the published garbage collection algorithms briefly and offer insight into the problems of designing a garbage collector for an object-oriented database. Then we will propose a general design approach for a garbage collector for Encore/Observer. Finally it will discuss future issues for garbage collection in a distributed environment.

1. Introduction

High level languages like Lisp, Prolog, and Smalltalk80 have efficient, automatic memory reclamation. These languages allocate many types of objects while a program is running. When any more object can not be allocated in the memory, the garbage collector collects the objects which are never used again and makes more memory space for other objects. On the other hand, in the C language and in Pascal a programmer has to handle memory reclamation by hand, that is, these languages provide users with functions for memory allocation and deallocation. However it is often dangerous to make the programmer responsible because a programmer might wrongly deallocate the memory space of an object though it is still used. Some objects are often referenced by more than two objects, therefore it is not necessarily true that an object is not needed when it is no longer referenced by some particular object. Smalltalk80 has a memory allocation function but does not have a deallocation function. In Smalltalk80, an object is reclaimed by the garbage collector after it is no longer referenced by any other objects.

We have a similar garbage problem in object-oriented database systems. In many current databases, it is possible to delete data items (i.e, records). In the case of object-oriented databases, if we delete one object (say x), there might be other objects that reference this x. The deletion of x can lead to dangling

references or to references that, when dereferenced, yield some undefined piece of storage. The wisdom of deletion semantics in an object-oriented database is questionable. In Encore/Observer system, we might prefer to adopt the semantics of garbage-collected languages, in which an object can not be deleted but, instead, references to objects are destroyed. When all references to an object have been removed, the garbage collector reclaims the storage occupied by that object.

Most of the published automatic garbage collection algorithms have been designed for main memory or virtual memory, though some are for the persistent languages. As we will see in later discussion, a persistent language does not necessarily have the same characteristics as an object-oriented database. We can not say that the best algorithm for (virtual) memory may be the best one for an object-oriented database because of differing performance characteristics in the two systems. For example, objects stored in an object-oriented database tend to have a long lifetime (e.g, permanent); whereas, objects in (virtual) memory are often temporary. Hence, garbage should be generated at a slower pace in an object-oriented database than in (virtual) memory. In addition, retrieving objects from an object-oriented database will take more time than reading an object in (virtual) memory.

2. Published Garbage Collection Algorithms

Many algorithms have been developed to perform automatic storage reclamation. The algorithms can be divided into two groups.

- scavenging algorithms
  scavenge memory by sweeping over it : these include Mark & Sweep and Copy-Compact algorithms.

- incremental algorithms
  perform a small portion of reclamation or garbage collection bookkeeping with each update. These include Baker's real-time,

Generation Scavenge, Reference Count and Deutch-Bobrow Reference Count algorithms.

Environments that utilize the scavenging method of automatic storage reclamation 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 the scavenging phase into small pieces so the pauses are shorter but more numerous. In environments that utilize an incremental method, each update takes a little extra time and, generally, there are no long pauses.

2.1 Mark & Sweep

This method was developed for high level programming languages like Lisp. A system written in these languages stops execution when it can not allocate the memory space for a new object, and starts to reclaim garbage objects. A garbage collector using a Mark & Sweep algorithm ([4], [5], [6], [9]) traverses all objects which can be reached from the root (stack or fixed address) and marks them. After that, it clears all memory space except the memory space used by marked objects.

2.2 Copy-Compact

Using a copy-compact algorithm ([13]), 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. 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 & 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.

Many high level languages like Lisp have been installed on workstations recently, and used for AI and program development environments that require real-time response. Mark & Sweep and Copy-Compact can not satisfy this requirement. Because of this, the following algorithms discribed in Sections 2.3 - 2.7 have been getting popular.

2.3 Baker's real-time

Baker developed a version of copy-compact that partitions objects into small groups so that the copying can be done gradually ([1], [9], [12]). In each partition a bounded number of objects are copied, thereby limiting the length of pauses in the program. Each time a new object is allocated, the pointers from k objects in copyspace are traversed. This allows the scavenging of memory to be performed incrementally. If a pointer refers to an object in oldspace, the object is copied into copyspace and the pointer is updated to refer to the copyspace location. Since pointers may still exist that refer to the object in the oldspace (after it has been moved to copyspace), a marker must be left in the old space objects that indicates that 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.

2.4 Generation-Scavenge

The Generation-Scavenge schemes ([1], [12]) assign ages to objects to reduce number of objects in memory that need to be considered when doing a copy. The age groups are determined by the algorithm implementor. A separate portion of memory is allocated for the objects of each age group. Each age group's root set consists of the objects in the age group that are pointed to by objects in older age groups. In virtual memory, the younger the object, the more likely it is to become garbage, so young spaces are scavenged more often than older spaces ([3]). The use of temporary variables in programming languages causes young objects to become reclaimable sooner. 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.

2.5 Reference Count

In this algorithm, each object has a reference counter which shows how many other objects are referencing it. Each time a reference is removed, the count is decremented. When the reference count goes to zero (which means that an object is no longer referenced), the garbage collector reclaims the object. Although the Reference Count scheme ([6], [10]) eliminates long pauses, it introduces considerable overhead into the normal operation of the system because it requires continual maintenance of the reference count field.

2.6 Deutsch-Bobrow Reference Count

This scheme ([2]) reduces the actual overhead of continually updating the reference count field as mentioned in Section 2.5. It is based on the observation

that the most frequent and dynamic object references are quite often transitory. If these references are eliminated, then almost all of the reference counting overhead will have been eliminated. Since most objects have a reference count of one, most of the space typically allocated to hold the counts is wasted. Deutch and Bobrow propose 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 addresses of the objects, but 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 can not be removed. Their scheme operates in the same way 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.

2.7 Atomic Garbage Collection

Some research ([15], [16]) focuses on atomic garbage collection used for transaction systems. An atomic garbage collection algorithm preserves the data consistency property. This means that since reliable distributed systems are required to maintain data consistency despite crashes (hardware crash , software crash), a crash during garbage collection should result in no loss of data. Kolodner ([15]) presents an atomic incremental garbage collector designed for the transaction-based language Argus. Kolodner's algorithm is based on a traditional stop-and-copy collector but is incremental. The main point of Kolodner's algorithm is that it allows for efficient crash recovery: in transaction systems, if the contents of non-volatile (disk) storage survives a crash, it is advantageous if it can be used as the starting point for recovery, instead of recovering entirely

from a log of modification records. Kolodner's collector allows recovery from non-volatile storage even if a crash occurs during collection. His approach divides the heap into volatile and stable areas. Objects are created in the volatile area. After an object becomes stable it is moved to the stable area at an appropriate time. The volatile area can be collected using incremental and/or generational collection. Therefore an incremental atomic garbage collector is used for only collecting the stable areas. This approach is based on the fact that an atomic garbage collection is more expensive than normal garbage collection. Why is it expensive ? Because the atomicity of an object is related to its transactions and those transactions should be recoverable from a system crash. Therefore the garbage collector must have an intimate relationship with the recovery system when the garbage collector reclaims the garbage.

3. Important considerations for designing on-disk garbage collection

We describe some important issues which should be carefully considered in designing on-disk garbage collection:

- The need for dealing with a huge amount of permanent data
  All of the objects which exist on the disk are permanent. The amount of data in the database is huge compared with the data in memory (The number of objects might exceed a million sometimes).

- Cost performance on both time and space
  Obviously on-disk garbage collection will take more time than that of a programming language because it must deal with huge amounts of data on the disk. As a result, a more efficient algorithm which requires less time for garbage collection is required. It raises the question of which algorithm is the best one with good performance both on time and space. Some algorithms require more space (e.c., Copy-compact, Generation Scavenge) and some require a long pause halting the entire system (e.c., Mark & Sweep, Copy-Compact), and therefore become expensive.

- Time requirement to deal with concurrent users
  Since in an object-oriented database system there exist multiple users interacting with the same objects concurrently, if we adopt the incremental garbage collection, the pause caused by update of the

reference count by Reference Count takes a longer time than that of the programming language because the new reference count must be broadcast to other users.

Then we have some questions immediately concerning how to approach garbage collecting a database file. First, which algorithm should we use, Mark & Sweep, or copying, or reference count ? Second, should we do a whole database at a time or can we do pieces ? Third, should we garbage collect off-line (batch) or on-line (incremental) ?

We will discuss these issues in the next section by analyzing each algorithm.

4. Our approach to adopt a published algorithm for the object-oriented database

Here we discuss which algorithm could be the candidate or suitable algorithm for on-disk garbage collection in an object-oriented database. We classified published algorithms into two groups shown below.

(Scavenging algorithms)
- Mark & Sweep
- Copy-Compact

(Incremental algorithms)
- Baker's real-time
- Generation Scavenge
- Reference Count
- Deutch-Bobrow Reference Count
- Kolodner's Atomic Incremental

At first, we eliminate Generation Scavenge from our study because all of the objects in the database are permanent. Performance benefits based on distinguishing between temporary and permanent data will not show much benefit in a persistent environment. Therefore, the assumption that allows Generation Scavenge to have a better performance than other algorithms (namely, that objects become garbage based on their age) does not necessarily hold for database objects. In addition to the above observation, the evolution of the garbage in an object-oriented database may be the reverse of that of the programming language.

In an object-oriented database world, the older data might be more likely to become garbage while in the programming language world the newer data is more likely to become garbage. Though we do not know how the age (in the sense of a generation) of the data in an object-oriented database can be defined, if we can define it, Generation Scavenge might be effective.

Second, Copy-compact and Baker's real-time can be eliminated from our study for several reasons. Both algorithms require twice as much disk space as is occupied by data. As mentioned before, we believe the rate of garbage growth on disk is comparatively lower than that of a high level programming language. Though garbage data could be born after some system operations (e.c., modify and delete references etc.), the rate of growth is much slower than that of temporary data in the programming language. In addition, since the database does not save temporary data at all, we believe that having another disk space only for garbage collection is too expensive in the view of the system design. Also copying huge amounts of data (almost the entire disk space) from disk to disk is too costly because almost all data on disk are reachable objects because of data permanency in the database.

Third, we omit Deutsch-Bobrow Reference Count from our study because the database does not store transitory references at all. We doubt the effectiveness of having two tables on disk which require a lot of space in proportion to the number of objects. In addition, searching and manipulating the reference count of each object from the huge table is too costly.

Fourth, we will leave out the discussion of whether the Atomic Incremental Garbage Collection is applicable to an object-oriented database or not for the following reason. We think the most important purpose of the Atomic Incremental Garbage Collector is to reorganize the heap to provide good paging performance. This reorganizing of the heap during garbage collection is similar

to the problem of clustering in object-oriented database systems in that the collector can increase locality of reference and reduce paging by moving objects that are referenced together to the same page by reorganizing the heap, making the collector move objects. In object-oriented databases, clustering occurs at two levels: (1) partitioning objects into areas, and (2) arranging objects within areas. If we adopt the scheme that a garbage collector should do the reclustering during a garbage collection, the Atomic Incremental Garbage Collection might be important. However, all data in object-oriented databases might be persistent data in some databases (see Section 5), therefore stable. If the atomic garbage collector for an object-oriented database has to deal with all data in the database, it will take a long time totally even if the reclustering processes can be incremental. In addition, we don't define clearly about the relation among the garbage collection, the process of reclustering the database and the recovery system. Our research work for this field should begin at this phase. Therefore we eliminate this scheme from our study at this time.

So far we have two basic algorithms left: Mark & Sweep and Reference Count. Mark & Sweep is an off-line collection, which requires long pauses and uses less space, but it can deal with all data in the database at a time, and is therefore simpler. On the other hand, Reference Count is an on-line collection, which is likely to be more difficult and imposes some additional overhead during collection, but is necessary for continuously running applications. Since we doubt the effectiveness of adopting Reference Count, because of the slow performance (therefore this might be more expensive) caused by a lot of updates of reference count during a real time system operation, we will leave this discussion as a future issue.

In view of the above observations, we decided to use the Mark & Sweep algorithm for the Encore/Observer system as described in next section.

5. Mark & Sweep General Design

We find the following problems to be solved when we attempt to apply Mark & Sweep to an object-oriented database: reachability of an object, traversing method to find reachable objects and cyclic object reachability.

5.1 Object Reachability

In general, the reachability of an object from the persistent root is used to decide whether the object is garbage or not. How is the reachability defined in an object-oriented database ? Can we use the same definition of the reachability as used in the programming language ? The answer is no. We explain why next.

Some object-oriented databases adopt the scheme that an object is connected to some persistent root objects. Then database objects have a persistent root object. Any object that is reachable from this persistent root is also persistent and reachable. This scheme is effective especially for querying. Some object-oriented databases adopt the concept of a type's extent: when a type is instantiated, the instance is kept in the collection of all instances of the type. Then these collections of objects are used in querying. But this scheme introduces a new problem ([11]): that all objects are reachable. Therefore reachability of an object (as a means of identifying garbage) must be defined some other way.

Our strategy for reachability in Encore is that reachability is defined by the availability of the link between two objects. That means we have the function to test the reachability (link availability) between one object and another object.

5.2 Traversing Method to Find the Reachable Objects

It is not easy to determine if the language-based depth-first search or breadth-first search marking schemes are effective when traversing objects on the disk, because sometimes such a search may cause more page faults and/or take a much longer time to check all objects (the search might get all objects into memory) and there is no effective way to determine which objects to page out. Therefore we should consider other traversing techniques for objects on disk which seem more suitable. We break traversal into two steps: one for defining reachability from the collection and the other for inter-object connections.

We adopt a traversing technique which is based on the segmentation of Encore. We will describe it here. Encore adopts a segmentation (clustering) technique to access the disk efficiently (in other words, to reduce the number of disk I/O). That is, by collecting the inter referencing objects as much as possible into one segment and then using the segment as the unit of pre-fetch. Encore speeds up the access reducing disk I/O ([11]) . In our Mark & Sweep scheme#1 and scheme#2 (to be presented in Section 5.4), the algorithms traverse all objects for marking on the segment access basis in the second loop in scheme#1 and in scheme#2. The algorithm checks every object in one segment by traversing every object it references and marks reachable objects because it takes advantage of clustering to reduce paging. However the algorithm stops traversing further with the next referenced objects' properties. If we continue looking at every object in one segment until we finish looking at all segments, eventually all reachable objects shall be marked. We believe this traversing technique is most suitable in Encore to look at all objects because it takes advantage of clustering to reduce paging.

5.3 Cyclic Object Reclamation

In programming languages, the Mark & Sweep algorithm sweeps over the entire memory space except the memory space used by marked objects to reclaim the cyclic objects. However this strategy is too costly for on-disk garbage collection because traversing over disk space which contains a large amount of data takes a long time. Therefore a new, less expensive strategy to reclaim the cyclic objects is required.

We will adopt a new algorithm described in the section 5.5 to detect cyclic objects, one which uses the type hierarchy information of Encore.

5.4 Two Mark & Sweep Techniques

In this section, we show two schemes and pseudocode for each for our Mark & Sweep algorithms. Scheme#1 checks object reachability by looking at the connection between an object and the persistent root at first, and second marking objects referenced by the object. Scheme#2 checks the reachability by using a reference count as a "mark". The object, which has the reference count of 0, is reclaimed. The difference between scheme#1 and scheme#2 is that: scheme#1 checks the reachability from the persistent root directly by using the function (which checks the connection between an object and the persistent root), on the other hand scheme#2 checks the reachability from the persistent root by using and modifying a reference count (if the count is 0, an object is not connected to the persistent root).

5.4.1 Mark & Sweep Scheme#1

This scheme checks whether the collection of instances of each type are connected to the persistent root. If an instance is connected, it is marked into its field which shows that this object is reachable from the persistent root. Then the

algorithm starts traversing each object to mark the other referenced objects. In this process, if the object which the algorithm is looking at is not reachable, it does not mark any objects referenced by this object because this object is not reachable. However, if this object is later referenced by an object that is connected to the persistent root (that is, it becomes reachable), this object also becomes reachable. As a result, all objects referenced by this object become reachable too: the algorithm recursively traverses all following referenced objects until it encounters the objects which are connected to the persistent root o null pointers (NIL).

```
Mark-Sweep-1 ()

------  Marking phase -------

        for every collection type object  c in the segment s
                for every instance i in collection  object c
                    if ( i is reachable)      -------  checked by macro
                            mark i with  "p"   -----  reachable from the persistent
                                                       root
      for every segment s in database
          for every  object  x in the segment s excluding a collection type object
                    if (x is marked with "p")    ------ x is reachable
                      then
                            for every instance i' referenced by this object x
                                if (i is not marked with "p")
                                    then  call follow_mark (i)
                                    else  mark this object with "1"

follow_mark  (p)
       mark this object with "1"
        for each instance w of property of p
              if (w is not marked with "p")
                        call follow_mark (w)


------    Sweeping  phase ------

       for each segment s in database
             delete all unmarked objects
```

5.4.2 Mark & Sweep Scheme #2

This scheme uses the reference count as a "mark." This means if the count is zero, the object is reclaimed in sweeping phase. Similar to Scheme#1, the algorithm looks at every object in one segment. If the object references another object, its count is incremented. After looking at all objects, the algorithm starts to modify the reference count by accessing every segment again. If the algorithm finds an object whose reference count is zero, it decrements the reference count of all objects referenced by this object by 1 because the object is not reachable from the persistent root, therefore the reference from this object is not valid. If the reference count of some object decremented by this algorithm becomes 0, the algorithm takes the same process recursively. The problem of this scheme is the delay or the long pauses caused by cascading mentioned above.

```
Mark-Sweep-2 ()

------  Marking phase -------

        for every segment s in database
            for every  object  x in the segment s
              increment the reference count of all objects by 1 referenced by this
              object x
        for every segment s in database
            for every  object  x in the segment s
                if ( reference count of x = 0)
                     call decrement_count (x)

decrement_count (x)
        for every property p of x
              if ( (reference count -1) of p = 0 )
                     then decrement_count (p)
                     else  decrement reference count of p by 1


------   Sweeping phase ------

        for each segment s in database
              delete all unmarked objects
```

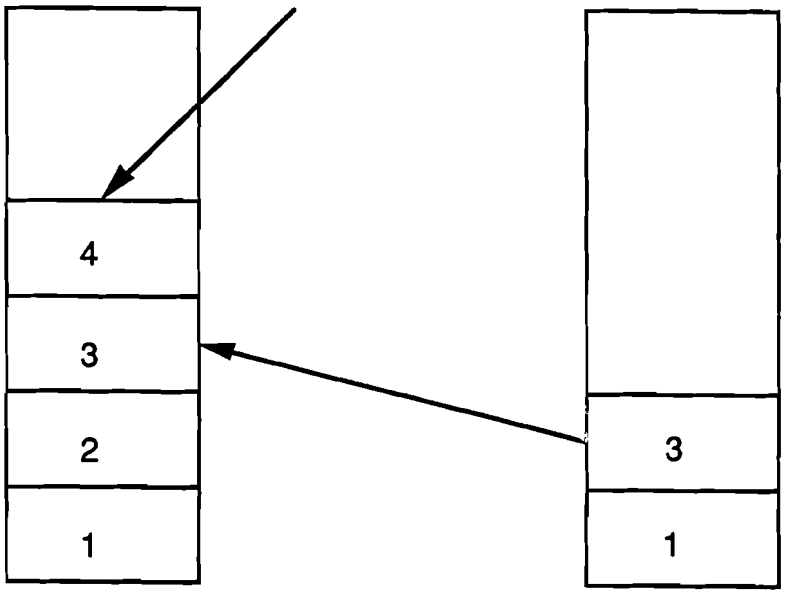
5.5 Algorithm to Detect Cyclic Object

In this section, we are going to describe an algorithm which detects cycles. In the database we might have cyclic objects which happen to be created by users. These objects obviously should be reclaimed automatically because finding them manually is almost impossible in a huge database. Encore maintains a type hierarchy which is based on the user schema. Specifically, the type hierarchy of Encore keeps information on a directed graph G(V, E), identifying which nodes represent each type and which edges represent references between two types. By understanding the Encore type hierarchy, we can know the possibility of cyclic objects in the system and which types could participate in a cycle. Our algorithm checks for this possibility and if a cycle exists among some types, it makes the list of these types. It traverses the database segment by segment and if it encounters some instance of the type in the above list, it starts checking from this instance, follows the instance of the next type which can be known by following the order of the list and repeats the same process until it checks each instance of each type in the list to determine if there is a cycle. If it finds a cycle, it stores the information of all instances which are in the cycle and reclaims them after checking all objects.

We describe the pseudocode of the cycle check algorithm below.

INPUT : Graph G (V, E)

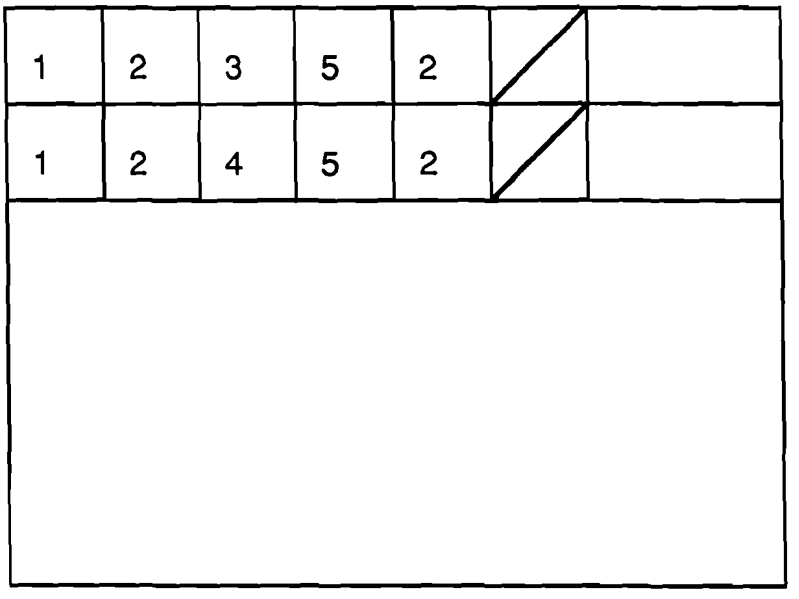
Data Structure :



search_table (stack)
: trace table of vertex

reverse_table (stack)
: is used to detect
all possible cycles
in DFS-VISIT

Figure 1. search_table and reverse_tabel



cycle_check_table
: possible ordered lists of type which could make a cycle

Figure 2. cycle-check table

```
DFS (G)
      for each vertex u ∈ V[G]
            π[u] <- NIL                                  π[u] : predecessor of u
      for each vertex u ∈ V[G]
            DFS-VISIT (u)
            clear_search_table (search_table)            : empties the stacks
DFS-VISIT (u)
      push (u, search_table)
      if |Adj [u] | >= 2                    Adj[u] : adjacency list of u
            push (u, reverse_table)
      for each v ∈ Adj [u]
            if (member (v, search_table) = TRUE)         : member check
                  then push (v, search_table)
                  register_list (search_table, cycle_check_table)
                                          : copy trace data to cycle_check_table
```

```
                reverse = pop (reverse_table)
                pop_list (v,  reverse,  search_table)  : pop list elements at a
                                                         time
                push (reverse,  reverse_table)
        else    π[v] = u
                DFS-VIST  (v)
pop (reverse_table)
```

This algorithm can list possible cycles into cycle_check_table from graph G. It finds out special cycles (the longest path cycles) which contain subcycles in the list. To do this, it eliminates the multiple equivalent cycles in the entire cycle. After that it checks the relationship among cycles by asking whether one cycle is a subcycle of the other or not.

That is, suppose that we had the cycles C1, C2, C3, -- Cn. If the length of the cycle path |Cj| > |Ci| (1<= i, j <= n), and Ci ∈ Cj, it eliminates all sub-cycles from the list.

For example in the graph (in Figure 3), we can find eight possible cycles.

C1 = 1-2-3-5-2
C2 = 1-2-4-5-2
C3 = 2-3-5-2
C4 = 2-4-5-2
C5 = 3-5-2-3
C6 = 4-5-2-4
C7 = 5-2-4-5
C8 = 5-2-3-5

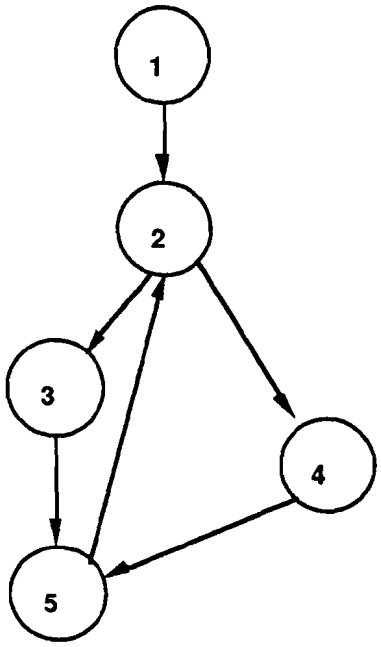


Figure 3. example

At first it eliminates the equivalence cycles from the list by looking for the following two conditions, whether |Ci| = |Cj| and whether the list contains the very same vertexes in the same order after eliminating the last vertex and sorting both lists by vertex number.

C3, C5 and C8 are equivalent. ---> eliminates C5, C8
C4, C6 and C7 are equivalent. ---> eliminates C6, C7

Since C3 is a subcycle of C1 and C4 is a subcycle of C2, only two cycles , C1 and C2 are left eventually.

C1 = 1-2-3-5-2
C2 = 1-2-4-5-2

6. Incremental garbage collection

In Section 5, we described a Mark & Sweep general design assuming that the garbage collector runs making the entire system stop. However stopping the system is sometimes not desirable in a system that requires continuous operation. If the system has a huge amount of data on the disk, the garbage collector might

cause a very long system halt. In this section, we will discuss the issue of how to make the object-oriented Mark & Sweep garbage collector incremental.

In a programming language the garbage collector and the executing program contend for CPU cycles. An incremental garbage collector allows its operations to be incremental with that of the program. In the object-oriented environment the garbage collector and the program run as separate programs so their execution is interleaved by the operating system. But, they do have to contend for object locks. So in the object-oriented database, the important resource is the object itself and the garbage collector must be able to deal with locked objects. In our model, we adopt the following new definition that the incremental garbage collector keeps running even if objects are locked. Why do we need this definition ? The reason is that if the algorithm has to wait for the commit of an object which is locked for a long time, not only does the total time of garbage collection increase but also the effect of waiting might corrupt the normal system operation.

Next we will explain the general algorithm of our incremental garbage collection. We regard the garbage collector as one of the multi-tasks in the system which runs under the control of the operating system. Therefore it runs with some time interval scheduled by the scheduler. The program's transaction causes objects to be locked. We have to deal with the effect of these operations which happen to occur during the marking phase of the garbage collection. We deal with these operations on the segment basis to make the table size for checking small. Before we explain the detailed algorithm, we should answer the general question of which scheme can be incremental, scheme#1 or scheme#2. The answer is both. Though we will describe pseudocode for each later on, we will explain the incremental algorithm with scheme#1 to give a general idea easily. But if you read the following explanation by using the reference count (0

or more than 1) as "mark" instead of using with 0 or 1 as "mark" to decide whether an object is garbage or not, you can make sure scheme#2 can be incremental. Now we will start to explain the algorithm.

If an object which the garbage collector is looking at is locked, it puts this object into a later_check_table and keeps on traversing in the segment. After checking all objects in the segment, the garbage collector starts checking the objects in a commit_table which stores such objects that committed during the garbage collection's segment traversal. After dealing with objects in the commit_table and deleting those objects (which have the same UID) from the later_check_table, the garbage collector moves to the next segment and continues checking until it finishes traversing all segments and checking all objects in the database. If there still exists some objects in the later_check_table, the garbage collector waits for the commitment of these objects until the time-out (time span is changeable by the parameter) occurs. When the time-out occurs, the garbage collector marks the objects referenced by these locked objects by using the old copy of the locked objects which are in the later_check_table. Some logical contradictions shown below might occur if a locked object commits and changes the state of some other objects to cause a logical contradiction after the garbage collector finishes checking and marking them.

(1) logical contradiction#1

An object, which the garbage collector interpreted as alive and marked with "1" when it was traversed, becomes dead at the point that the garbage collector finishes marking all objects.

(2) logical contradiction#2

An object, which the garbage collector unmarked with "0" (dead) when it was traversed, becomes alive (reachable) at the point that the garbage collector finished marking all objects.

Since an object of case (1) will be reclaimed in the next cycle of the garbage collection, we don't have to care about this.

We should consider following three different situations which arise in the second case.

- situation#1

  The objects which are not marked because of a lock should be made be alive again after the garbage collector finishes sweeping. In some object-oriented databases (such as Encore), to guarantee the processing of the other application programs like a query which might use the locked objects, they have copy (objects before the lock) of the locked objects in the memory. Then, since the application has a copy of the locked objects in the memory, it is possible to make the object alive again by saving the memory copy into the disk.

- situation#2

  New objects should be stored into the disk after the garbage collector finishes sweeping. This is possible since all new objects are created in the memory of the application.

- situation#3

  If an application program makes the new reference to an object that is on disk but not in memory, this object can not be made alive again since there is no copy in the memory (for example, if an object copies a reference to another object without loading the new referenced object into memory). To avoid this, during the garbage collection mode, the application program should have a memory copy of the referenced object when it makes the new reference. Then the garbage collector can make new referenced objects alive.

In addition to the above, during the sweeping phase, the garbage collector protects the unlocked objects from being locked by an application program by forcing the transaction to wait so as not to cause another logical contradiction.

Now we will show the pseudocodes in the following.

```
Incremental-Mark-Sweep-1 ()

------  Marking phase -------

        for every collection type object  c in the segment s
                for every instance i of collection type object c
                    if ( i is reachable)        -------  checked by macro
                            mark i with  "p"    ----  reachable from the persistent
                                                      root
    for every segment s in database
            for every  object  x in the segment s excluding a collection type object
                    if (x is locked)                  ----  object is locked ?
                            then   push (x, later_check_table, "F") ---  object is
                                                                         locked
                            else   if (x is marked with "p")     ------ not locked,
                                                                        x is reachable ?
                                   then
                                         for every instance i' referenced by this
                                                object
                                           if (i is locked)
                                                then push (i, later_check_table, "
                                                                        S")
                                               else  if (i is not marked with "p")
                                                       then  call follow_mark (i)
                                                       else  mark this object with
                                                                  "1"
        while (there is an object in commit_table)
             N=pop (commit_table)              -- pop any object from the table
             pop (N,  later_check_table)        --  pop the special object
             case  (the flag of N)
                    F:          if (N is marked with "p")     ------ not locked,
                                                                       N is reachable ?
                                then
                                      for every instance i' referenced by this
                                             object
                                        if (i is locked)
                                             then push (i, later_check_table,
                                                                       "S")
                                             else  if (i is not marked with "p")
                                                     then  call follow_mark (i)
                                                     else  mark this object with
                                                                "1"
                           S:          if (N is not marked with "p")
                                                     then  call follow_mark (N)
```

```
                                                    else  mark this object with
                                                            "1"
                        T:          if (N is not marked with "p")
                                            call follow_mark (N)
if (time out occurs)                                    --- wait for the time out
    then
            while (there is an object in commit_table)
                  N=pop (commit_table)                  -- pop any object from the
                                                          table
                   pop (N,  later_check_table)          -- pop the special object
                  case  (the flag of N)
                      F:          if (N is marked with "p")     ------ not locked,
                                                                    N is reachable ?
                                  then
                                        for every instance i' referenced by this
                                              object
                                          if (i is locked)
                                               then push (i, later_check_table,
                                                                    "S")
                                              else  if (i is not marked with "p")
                                                      then  call follow_mark (i)
                                                      else  mark this object with
                                                              "1"
                        S:          if (N is not marked with "p")
                                                      then  call follow_mark (N)
                                                      else  mark this object with
                                                              "1"
                        T:          if (N is not marked with "p")
                                            call follow_mark (N)

           if (there is a object in later_check_table)
              then by using a old copy in memory,       -- time out occur
                     every object Y in the later_check_table
                case (the flag of Y)
                   F:          if (Y is marked with "p")     ------ not locked,
                                                                    Y is reachable ?
                               then
                                     for every instance i' referenced by this
                                           object
                                       if (i is not marked with "p")
                                                      then  call follow_mark (i)
                                                      else  mark this object with
                                                              "1"
                     S:          if (Y is not marked with "p")
                                                      then  call follow_mark (Y)
                                                      else  mark this object with
                                                              "1"
                     T:          if (Y is not marked with "p")
                                            call follow_mark (Y)

    else  while (there is an object in later_check_table)
             while (there is an object in commit_table)
                N=pop (commit_table)                    -- pop any object from the
                                                          table
```

```
                pop (N, later_check_table)        -- pop the special object
                case (the flag of N)
                    F:          if (N is marked with "p")    ------ not locked,
                                                                 N is reachable ?
                                then
                                      for every instance i' referenced by this
                                            object
                                        if (i is locked)
                                            then push (i, later_check_table,
                                                                    "S")
                                          else  if (i is not marked with "p")
                                                  then  call follow_mark (i)
                                                  else  mark this object with
                                                          "1"
                    S:          if (N is not marked with "p")
                                                  then  call follow_mark (N)
                                                  else  mark this object with
                                                          "1"
                    T:          if (N is not marked with "p")
                                          call follow_mark (N)


follow_mark (p)

        mark this object with "1"
         for each instance w of the property of p,
                    if (w is locked)
                        then   if ( w is not in the later_check_table)
                                        push (w, later_check_table, "T")
                               else   if (instance w are not marked with "p")
                                        call follow_mark (w)

------   Sweeping phase ------
        prohibit new lock by an application program
        delete all unmarked objects
        copy all objects in memory into disk at one time
        allow new lock by an application program


Incremental-Mark-Sweep-2 ()

------  Marking phase -------

        for every segment s in database
            for every  object  x in the segment s
                  if (x is locked)
                       then push (x, later_chek_table, "F")
                        else increment the reference count of all objects by 1 referenced
                        by this object x
           while (there is an object in commit_table)
              N=pop  (commit_table)          -- pop any object from the table
                pop (N, later_check_table)          -- pop the special object
                table_data_pocess (N)
```

```
      for every segment s in database
          for every  object  x in the segment s
                if (x is locked)
                     then push (x, later_check_table, "S")
                      else if ( reference count of x = 0)
                           call decrement_count (x)
          while (there is an object in commit_table)
            N=pop  (commit_table)          -- pop any object from the table
              pop (N, later_check_table)          -- pop the special object
              table_data_pocess (N)

      if (time-out occurs)
           then
            while (there is an object in commit_table)
                    N=pop (commit_table)          -- pop any object from the table
                      table_data_pocess (N)
            if (there is a object in later_check_table)
                   then    by using an old copy in memory,
                           every object N in the later_check_tbale
                             table_data_pocess (N)

           else
            while (there is an object in later_check_table)
                    while (there is an object in commit_table)
                             N=pop (commit_table)  -- pop any object from the table
                            table_data_pocess (N)

decrement_count (x)
       for every property p of x
             if (p is locked)
                    then  push (p, later_check_table, "T")
                    else  if ( (reference count -1) of p = 0 )
                          then decrement_count (p)
                          else  decrement reference count of p by 1

table_data_pocess  (N)
             case (the flag of N)
                       F:  increment the reference count of all objects by 1
                                                  referenced by this object N
                       S:  if ( reference count of N = 0)
                                 call decrement_count (N)
                       T:  if ( (reference count -1) of N = 0 )
                              then decrement_count (N)
                              else  decrement reference count of N by 1


------   Sweeping phase ------

       prohibit new lock by an application program
       delete all unmarked objects
       copy all objects in memory into disk at one time
       allow new lock by an application program
```

Last we should note that the described garbage collector can be equipped with the important feature of compaction and reclustering after sweeping over the garbage objects. This is an important aspect in an object-oriented database.

6.1 Collecting Garbage Within a Transaction

Next we will discuss about the reclamation of the objects which are created within and left unreferenced at the end of a transaction. Supposed that during a transaction we had the case that objects A, B, C are created and referenced by an object X at first, however finally only A is referenced by X. At commit time, A, B, C are stored on the disk. Obviously B and C are garbage. So if we can avoid creating obvious garbage on the disk, we should. To avoid creating this inter-transaction garbage, before commit time another local garbage collector (we call this "auxiliary garbage collector") which resides in each application checks the reachability of all objects and collects garbage. We explain this process more.

In each program space, we have the table which keeps the status information of all objects (OLD, NEW, UPDATED etc,.): O - old object (no change occurred), N - newly created object, U - updated object.

By checking the reachability of all new objects with the algorithm of scheme#2 used in the marking phase and eliminating some of them which are garbage from this table, we can avoid creating some garbage on the disk (not by sending these to the server).

7. Future issues

Implementation and performance on Encore/Observer are urgent future issues in this research field. One paper ([13]) shows performance data of garbage collection. However they are data for the programming language not for

the database system. We should measure the performance of the object-oriented database system and check whether we need some improvement or not.

We should collect more performance data (especially the effect of delays caused by updates of a reference count during system operation) of Reference Count so that we can decide whether it is possible to adopt this scheme for object-oriented database systems or not.

We should discuss whether the Atomic Incremental Garbage Collection is applicable to an object-oriented database or not since reclustering the database after the sweeping phase is an important issue in an object-oriented database world in terms of the efficient use of the disk.

Distributed Garbage collection is a more interesting research topic since we have to deal with the problem of synchronization among distributed databases during garbage collection and the problem of how to handle the reclamation of cyclic data among these systems efficiently and so on. This issue might be complicated and difficult to handle.

Recent research work on the garbage collection in the programming language mainly focuses on how the garbage collector can divide the objects into a couple of groups based on the age (generation) to reduce the time for copying. This emphasis on age-based garbage collection may indicate that Generation Scavenge might be useful if it can be adopted to an object-oriented database world. This research should begin by defining the concept of the generation of the objects, that is, what is generation based on in an object-oriented database and how can the age of objects be classified.

8. Acknowledgement

I would like to thank my advisor Stanley Zdonik for providing motivation and encouragement to work for this project. I wish many thanks to Page Elmore for helping me to solve a lot of hard problems and for giving constructive advice.

References

[1] David Michel Unger, The design and Evaluation of a high Performance Smalltalk System (An ACM Distinguished Dissertation 1986)

[2] L.P. Deutch and D.G. Bobrow, An Efficient Incremental Automatic Garbage Collector, Comm. of the ACM19, 9 (September 1976), 522-576

[3] H. Lieberman and C. Hewitt, A Real-Time Garbage Collector Based on the Lifetimes of Objects, Comm. of the ACM 26,6 (June 1983), 419-429

[4] J.K. Foderaro and R.J. Fateman, Characterization of VAX Macsyma, Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, Berkeley, CA, 1981 , 14-19

[5] J. Cohen, Garbage Collection of Linked Data Structures, ACM Computing Surveys 13, 3 (September 1981), 341-367

[6] T.A. Standish, Data Structure Techniques, Addison-Wesley, Reading, MA, 1980

[7] H. Schorr and W.M. Waite, An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, Comm. of the ACM 10,8 (August 1967), 501-506

[8] G.E. Collins, A Method for Overlapping and Erasure of Lists, Comm. of the ACM3, 12 (DEC 1960), 655-657

[9] S. Ballard and S. Shirron, The Design and Implementation of VAX/Smalltalk80, in Smalltalk-80, Bits of History, Words of Advice, G. Kransner(editor), Addison Wesley, 1983, 127-150

[10] D. Knuth, The Art of Computer Programming, Volume 1, Addison Wesley, Reading, MA, 1973

[11] Zdonik and Maierr, Readings in Object-Oriented Database Systems, 302-304, 325-326, 273-278, Morgan Kaufmann

[12] David Unger, Generation Scavenging : A Non-disruptive High Performance Storage Reclamation Algorithm, ACM Software Eng. Notes/SIGPLAN Notices Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984, 157-167

[13] Margaret H. Butler, Persistent LISP: Storing Interobject References in a Database, PhD thesis , Report No. UCB/CSD 88/401 Nov. 1987

[14] Peter B. Bishop, Computer Systems With a Very Large Address Space and Garbage Collection, MIT/LCS/TR-178

[15] Elliot K. Kolodner, Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap, MIT/LCS, Proceedings of the Fourth International Conference on Persistence Object Systems, Sept. 1990

[16] David L. Detlefs, Position Paper: Concurrent, Atomic Garbage Collection, ECOOP/OPSLA Oct.1990