Garbage Collection in Object Oriented Databases Optimization of Unreachable Objects detection

Myriam Lamolle, Marc Gonzalez, Thierry Millan, Pierre Bazex

▶ **To cite this version:**

Myriam Lamolle, Marc Gonzalez, Thierry Millan, Pierre Bazex. Garbage Collection in Object Oriented Databases Optimization of Unreachable Objects detection. Workshop on Computer Science and Information Technologies, Dec 2000, UFA, Russia. hal-03580875

HAL Id: hal-03580875

https://hal.science/hal-03580875

Submitted on 18 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Garbage Collection in Object Oriented Databases Optimization of Unreachable Objects detection

Myriam Lamolle
IRIT-CNRS (UMR 5055),
Université Paul Sabatier
Toulouse, France
lamolle@irit.fr

Marc Gonzalez
IRIT-CNRS (UMR 5055),
Université Paul Sabatier
Toulouse, France
marc.gonzalez@cisi.cnes.fr

Thierry Millan
IRIT-CNRS (UMR 5055),
Université Paul Sabatier
Toulouse, France
millan@irit.fr

Pierre Bazex
IRIT-CNRS (UMR 5055),
Université Paul Sabatier
Toulouse, France
bazex@irit.fr

Abstract

This article presents new garbage collection algorithms to improve memory management in Object Oriented Databases (OODB). A toward garbage collector has highlighted a lack in the way of optimisation about the detection and reallocation of free memory space. In the context of OODB, the essential idea is the detection of unreachable objects. We propose a solution based on the graph theory. In fact, we claim the main problem is the graph tracing to detect unreachable object or unreachable cycle. The cycle can be replaced by the large cell which has the same characteristics as a simple object. To begin the process of garbage collection, we use a determinist method to locate unreachable objects at the end of a transaction. Moreover, the garbage collector reclaims the memory space of unreachable objects. To implement these concepts, we use a reference counting and a strongly connected components table. Some examples supports the concept of our algorithms, in particular the special cases.

1. Introduction

Managing memory is still a main issue during application runtime because new data representations (sound, picture, movie, etc.) and new capabilities of communication increase data volume used by applications. Moreover, these data evolve very quickly. Thus, some data become obsolete after some months, and even some days.

The first garbage collector (GC) forced applications to stop in order to delete the false data and to reallocate the memory. This however, is not reasonably good. New developments try to overcome it by running GC in the background. In the database context, LISP GC algorithms are used, but they are inadequate here. However, in the relational database context, objects (rows) are explicitly deleted. Two main ways are used that *is reference counting* and *mark and sweep* [1]. But, in the specific context of Object Oriented DataBases (OODB), they must be optimized because new constraints (concurrent access, great data volume, transaction, data on disk, etc.) must be taken into account. Also, we suggest another algorithm for a centralized GC and memory reallocation.

In the following section, we present the main issues to manage memory and the different GC in the OODB context. In the third section, we propose a general solution optimizing the work of the GC by using the concept of graph and of graph's path. The fourth section exposes in detail the particular case, then, the particular case of Strongly Connected Components (SCC). In the fifth section, we describe the suggested algorithm, a reference counting and a table of SCC to implement the GC. At the end, we conclude this paper by future works to optimize in the sixth section.

2. Garbage Collection Background

At present, DataBases (DB) manage more and more complex objects like multimedia data. Moreover, it is easily possible to access to distant DS through network (internet, intranet, etc.). So, several users can simultaneously work on the same document, implying different versions of this document. On the other hand, data become very quickly obsolete. Obsolete data must be

detected by the GC to recover memory space. The memory space can be freed and be reallocated to new objects. However, the object's size is variable during his lifecycle. The first issue to solve the reorganization of memory to avoid scattering of object in memory. This is baneful to system performance. Moreover, grouping together objects allows to obtain a more important free continuous memory space. So, this space can contain new very large objects.

A GC is a system which detects if the data are used and automatically deletes the unused data. It distinguishes between reachable or unreachable objects. It is used by OODB Manager (OODBM) because the integrity of object memory can be violated either by the explicit user's deletion or the implicit deletion of program.

An OODBM, to recover memory or disk space, have two different processes:

- Collecting unreachable objects in memory; it is the main method to stop processes latching memory,
- Collecting unreachable objects in disk; it is the main method to stop process latching disk space.

These two types of collection are based on reachable data concepts:

- objects in memory are reachable if they are reachable from other objects in the scope,
- objects of disk are reachable from roots or active process.

It seems essential to implement an automatic process respectively named GC and memory reallocator. Presently, two kinds of GC are used [2]. The first one uses reference counting [3]. The second uses the mark [4] in which the objects of graph are tracing from roots. Then the GC destroys all unmarked data because the GC identifies unmarked data as unreachable objects.

2.1 Reference counting

Here, the number of references to each object is updated. A GC based on reference counting have two main strengths:

- it is dynamic because the memory space is recoverable as soon as the counter is equal to zero.
- it is incremental, that is the changes of the graph are taken into account as one goes along.

However, three weaknesses exist:

- the reference counting have a limit; a reference counting of great size is needed to be able to count the number of referenced objects,
- two counters are modified during an allocation; the one of indicated object and the other of referenced object whose the value is tested,
- the unreachable cycle of objects can never be identified.

2.2 The Mark-Sweep Garbage Collection

The principle of these garbage collectors consist in mark objects sequentially from a root to leaves. Marked objects are considered *alive* (i.e. reachable), others objects are *dead* (i.e. unreachable). This principle has fathered two types of garbage collectors:

- **Mark and Sweep** [4]: in the first phase, all data, which are reachable objects are marked. Then, in the second phase, unmarked data is deleted.
- **Copy** [5]: the reachable objects are copied one by one into a new memory address.

2.2.1. The classical Mark-Sweep [4]

The former, after having marked objects as explained above, recycles in a second step unmarked objects (i.e. recovery of the memory space taken by these objects). For example, namely the database at time T (Figure 1) with marked objects (A, B C from R1), with unmarked objects at this time (F will be by R2 in the future), and others that it will be never (D and E). The second step will consist in recycle memory space busy by objects D and E.

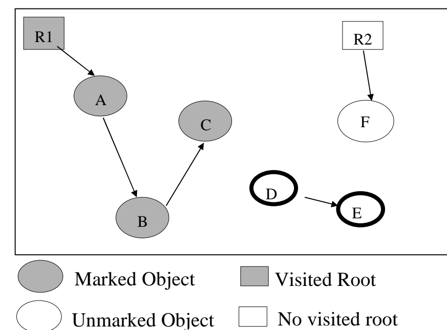


Figure 1 Marked objects in a database at time T

2.2.2. The Copying [5]

This GC merges the two marking and sweeping phases. In this algorithm, GC divides the memory in two parts. At the beginning, the first space contains DB' objects. The other space is empty. Here also, objects are covered from a root to leaves. Each met object is copied in the second memory space, some to the continuation of others. No copied objects are unreachable objects. Once the tracing of objects is finished, GC destroys all objects of first memory space. The second memory space becomes the current DB on which will work the GC to its next execution, and thus of continuation.
For example, the figure 2 shows the state of memory spaces of a DB at the instant T-1 (zone A) and at the instant T (zone B). At the instant T-1, the DB gets, in its first memory space (zone A), marked objects because they

have been reached by roots (grey objects on the figure). The GC copies marked objects in the second memory space (zone B) at this instant T. Then GC has to destroy the content of the zone A.

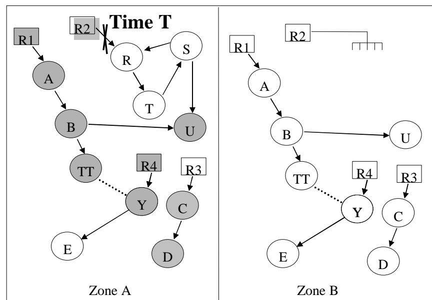


Figure 2 Two half memory spaces for DB

An optimisation of these algorithms exists in the generational garbage collection (half-space). Here, the most recently created objects have the most chance of becoming unreachable objects. The memory is not therefore divided in two spaces but in a certain number of generations. Each sweeping of the memory is more rapid because only a part of the memory is cleaned.

2.2.3. Garbage Collection in the Database context

However, these different algorithms are not viable in the context of DB for below reasons:

- large volume of data (penalisation of users depending on the work of GC)
- effect of swap that exists between the memory and the disc (due to difficulty of the immediate or deferred update)
- management of transactions: an object becoming unreachable during the transaction can again become reachable by the principle of rollback (attachment/detachment [6]). Some markers-sweepers have found a solution by posing reading or writing locks on objects.
- concurrent accesses management that can entail conflicts between transactions, and conflicts between transaction and GC,
- knowing the objects complete structure is very hard (for example, no possibility to use the concept of free memory space by a specific instruction of programming language).

To make viable algorithms of mark-and-sweep in the DB context, Mulatero [7] has introduced the principle of backward pointer. By leaving from a given object, the GC ascends by bottom-up transitivity from object to object until a possible root. Two cases are then possible.

In the figure 3.a, the greyed subgraph is reachable from the object O3 to root R1.

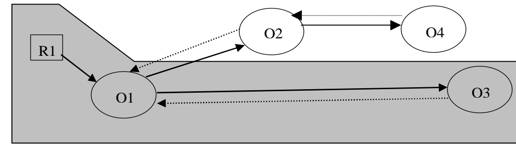


Figure 3.a Case of reachable subgraph

In the figure 3.b, the greyed subgraph is unreachable from object O2 because the GC ascends to object O4 and cannot go to another object or to root.

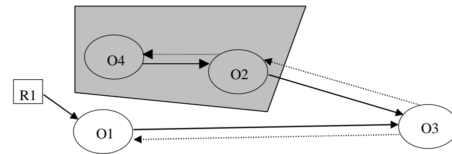


Figure 3.b Case of unreachable subgraph

Here, the GC and the transaction work in opposed senses therefore conflict between the two**.** In addition, it is incremental therefore the GC does not cover all the DB to recuperate the memory space. But management of backward pointer is expensive in memory space. Moreover, the choice of an object as the starting point of the algorithm is made manner no determinist. It is therefore imperative to cover all the DB before to know if all unreachable objects have well been collected.

Taking into account these different algorithms, we envisaged to put in place a GC in the context of DB such that:

- capable to choose an object of manner determinist,
- incremental and executing in parallel to transactions,
- automatic,
- capable to manage the table of indexes according to the reorganisation of objects stated in memory.

3. Garbage Collection algorithm – Focus of this paper

The principle of our GC is to leave from *dereferenced* objects. It is therefore necessary to use the concept of transaction. Indeed, when a transaction is validated, a given object can have lost one or several *incoming* references. For example, the figure 4 shows a partial view of DB.

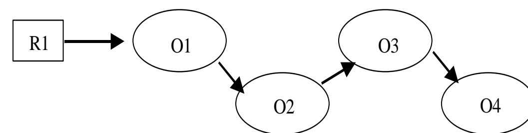


Figure 4 Partial view of DB before transaction T1

During the transaction T1, the link from object O1 to object O2 is going to be cut. The figure 5 shows the new state of DB.

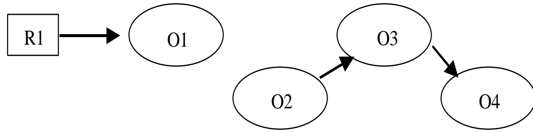


Figure 5 Partial view of DB after transaction T1

The underlying idea is to process all objects that lose their incoming reference(-s) (object O2 in figure 5) then their following objects (object O3 in figure 5). The GC works, in fact, by descending transitivity. This technique allows us to no use the notion of backward pointer.
In addition, this algorithm has particularity to be incremental to the level objects. The GC chooses objects of determinist manner, i.e. susceptible objects to be unreachable (loss their incoming references).
Finally, it includes the spatial locality notion of objects (processing of next objects).

4. Study of the different cases of unreachable objects detection

4.1 Basic case

It is the presented trivial case above (figure 4 and figure 5) and that serves as basis to our algorithm of garbage collection. When GC detects an unreachable object Oi (its number of incoming references is 0), the GC decreases the incoming references number of the following objects of Oi. Then GC cuts the link between Oi and its following objects and frees the memory space allocated to object Oi.

4.2 Several objects lost incoming references

Several objects, at the end of the transaction, can lose incoming references. Then, it is necessary to process all branches in parallel.
The GC is going to simulate the execution of several GC corresponding to each action to realise on each branch.

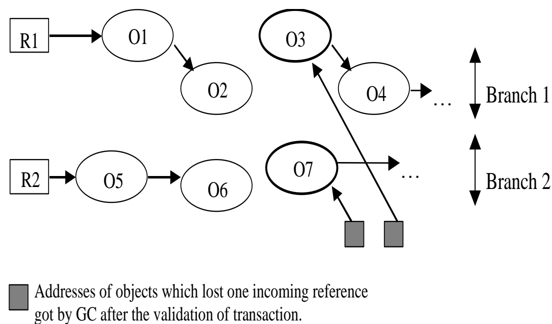


Figure 6 Parallelisation during GC's processing

The figure 6 shows the GC is going to process simultaneously object O7 and object O3, then by descending transitivity, the following objects of O3 and of O7.

4.3 Object having one incoming reference

However, the GC can find an object that again possesses at least one incoming reference after having processed the preceding object.

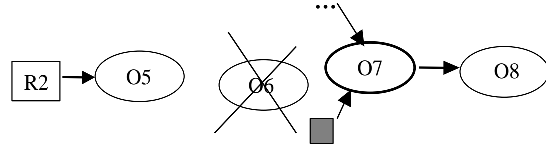


Figure 7 Action on object O7 after dereferencing from object O6

The figure 7 shows the deletion of object O6 by the GC. The GC processes then the case of object O7. This last possesses another incoming reference. A priori, the GC cannot therefore conclude if object O7 is reachable or no. We are going to propose solutions envisaged according to the different cases below.

4.3.1 Detection of one root

The figure 8 exposes the case where object O7 is connected indirectly to a root (R1).

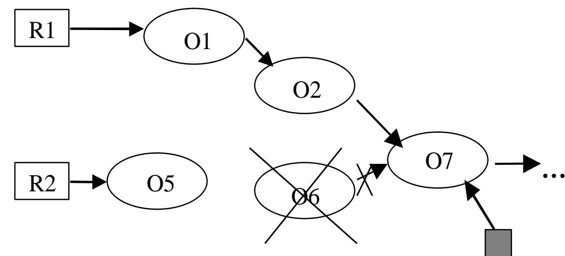


Figure 8 Case of reachable branch

The GC concludes thanks to the detection of a root that the object O7 is reachable. The GC stops its action on this branch.

4.3.2 No root

The GC does not detect connected root directly or no to the processed object.

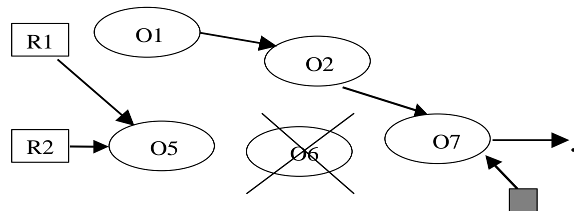


Figure 9 Case of unreachable branch

The figure 9 shows that the root R1 has been assigned to the object O5 during the transaction. During of processing of object O7 by GC, no root is detected. But the value of incoming references counting of object O7 is 1. Therefore, the GC stops its action on object O7 and its next objects.

Remark: in this example (figure 9), an another GC will begin also a processing from the object O1 (because O1 is unreachable) what will destroy object O7 and its following objects.

4.3.3 Detection of cycle

The GC cannot conclude directly if the object is reachable or not when a cycle exists.
For example, the objects O2, O3 and O7 on the figure 10 form a cycle.
We are then confronted with a mathematical problem concerning a Strongly Connected Components (SCC) of graph [8].
It is necessary to adapt algorithms proposed by [9, 10] to find SCC of a graph in the context of DB. The SCC is determined in the following manner :

- constitution of the list of all direct and indirect following objects of starting object (O7 in the figure 10)
- For all these following objects (O3 and O2 in the figure 10), verification by descending transitivity of a comeback on the initial object that determines the membership or no to the SCC.

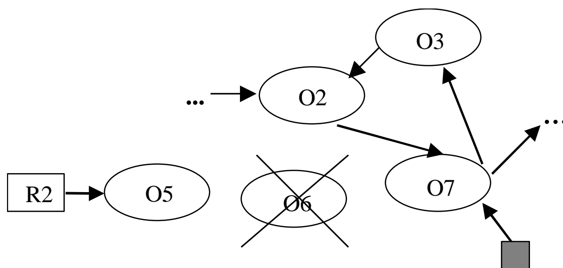


Figure 10 Detection of cycle

In figure 10, the SCC {O7, O3, O2} will be replaced by an object called **large cell** possessing the same characteristics that the other objects. It can therefore also possess one or several incoming references. The internal links of large cell are temporary deleted.
Then, figure 10 becomes

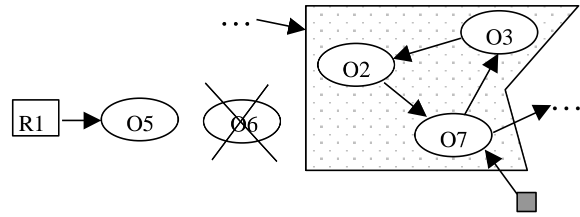


Figure 11 reduction of graph by SCC

Two configurations are then foreseeable.
Configuration1: The large cell does not possess incoming reference. The GC destroys all internal links, then internal objects of the large cell (figure 12) that is considered unreachable. The GC processes the next object of the large
cell.

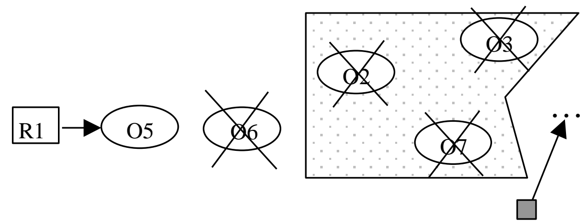


Figure 12 Deletion of links and objects inside the SCC

Configuration2: The large cell possesses at least a incoming (figure 11). The GC does not know if the large cell is alive or not. The GC stops its processing about this branch (cf. section IV.3.1 and section IV.3.2). The internal links of SCC are restored.

N.B: if the SCC is unreachable, it is detected in the future by another GC

V. General Algorithm of GC

V.1 Summary example

- The GC visits an object without incoming reference. The GC has cut the link between object O5 and object O6. The GC detects the unreachable object O6 (figure 13) because the number of 06's incoming references is 0 (cf. section IV.1). The GC eliminates object O6 then pass to the next object O7.

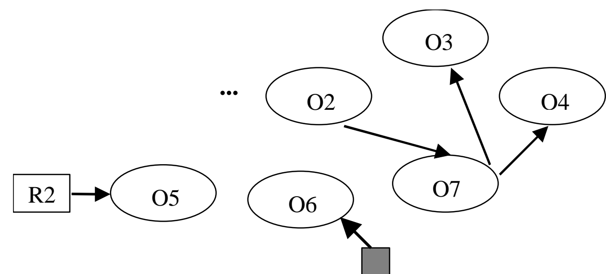


Figure 13 Processing about object O6

- The GC visits an object with an incoming reference. Determination of the SCC. In the figure 14, there is no SCC.

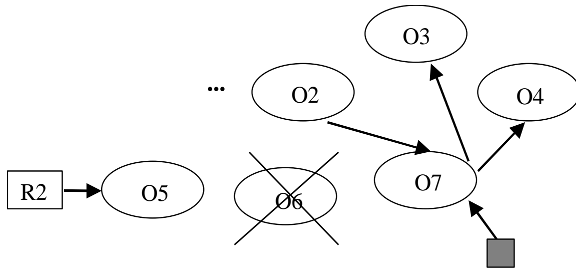


Figure 14 Processing about object O7

The GC has now to determine either the presence of a root in another branch to which object O7 belongs (section IV.3.1) or the absence of a root (section IV.3.2).
If a SCC had been detected, the GC would have reduced this last in large cell and would have made the processing of section IV.3.3.

V.2 Algorithms of Garbage collection

V.2.1 Main Algorithm

Begin
If no incoming reference **Then**
Unreachable object and treatment of following objects (cf. section IV.1)
Else
/* perhaps existing cycle */
If *cycle* **Then**
If SCC have incoming reference **Then**
Stop the process of this branch (cf. section IV.3.3.Configuration2)
Else
Deletion of all objects of SCC (cf. section IV.3.3.Configuration1)
EndIf
Else
No conclusion (cf. section IV.3.1 or section IV.3.2)
EndIf
EndIf
End

V.2.2 Algorithm of function *Cycle* used by the main algorithm

Begin
Put in a list *next_object* all no redundant following objects of the object O (including himself)
If there exists a path from O to O **Then**
Cycle ← O
For all element E to *next_objects* **Do**
If there exists a path from E to O **Then**
Cycle ← Cycle + E
EndIf
EndFor
Deletion of internal links and of internal objects of Cycle
return (true)
Else
return (false)
EndIf
End

V.3. Implementation of Algorithms

To implement these algorithms in the context of DB, we need two tools, namely:

- A incoming reference counting: it is incremented one by the transaction to each new incoming reference. It is decreased one by the GC when the object loses an incoming reference.
- A SCC table:

Oi	…	Oj	…
		Oi	

← Next objects of Oi

Next objects of Oj

Figure 15 : example of SCC table

In our example of table (figure 15), the object Oi is the starting point of the SCC. The first range represents all its indirect and direct next objects. Columns memorize all indirect and direct next objects of objects of the first range. If the object Oi appears in the column then the object starting point of the first range (object Oj in Figure 15) belongs to the SCC.

We tested these algorithms with O_2 OODB. We remarked a good treatment in the trivial cases and in the case of cycles having little SCC.

V.4. Algorithm Proof and Complexity

The justification of this algorithm is made by the Tarjan's algorithm [FRO93] whose we have made an adaptation for DB. This algorithm differs on the one hand, by the simplification of research of SCC since we are interested only in one SCC at once and not to a complete research of all SCCs of graph, and on the other hand to the replacement of the pile by a list.

Two properties that are very important and reused in the proof of the Tarjan's algorithm are:

- ∀ X, Y, 2 summits ∈to the same SCC. If ∃ a summit Z ∈ a path from X to Y Then Z ∈ to this same SCC.
- ∀ X, Y, 2 summits / ∃ a path from X to Y. If in a in-depth tracing, one marks X Then Y will be marked also.

The algorithm complexity is calculated according to the complexity level of the SCC table.

V.5 Examples

Our algorithm, by using reference counting, allows it to end its implementation, Which we will see in the following examples.

V.5.1 Example 1

Let's explore the graph without cycle (figure 16):

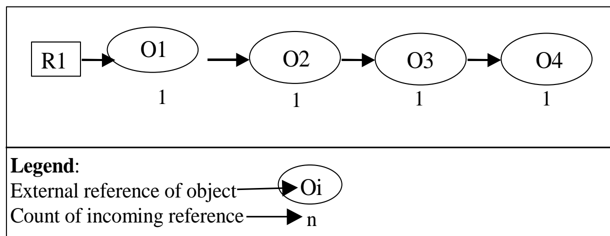


Figure 16 Graph representing a partial view of a DB

If the transaction commits to be validated by rendering the database in the following state (figure 17):

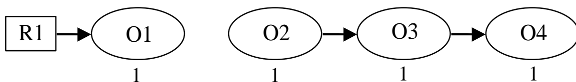


Figure 17 Deletion of link during validated transaction

Then the GC receives the object's address that has lost its incoming reference (figure 18).

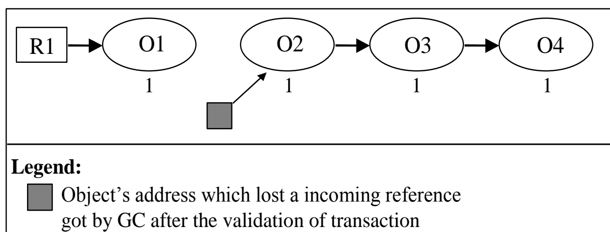


Figure 18 Transmission of object's address to GC

The GC decreases the reference counting and then we obtain the figure 19.

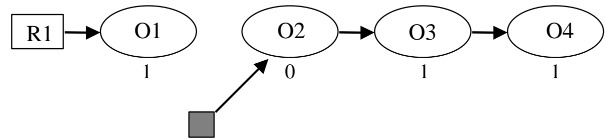


Figure 19 Modification of the value of incoming references counting of object

The incoming references counting being null, this object is unreachable ; the GC eliminates all *outcoming* references of this object then will visit all objects that lose one or several incoming references.

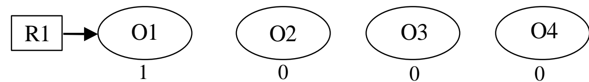


Figure 20 Transmission of address of next object

At the end of the GC process, we obtain the Figure 21

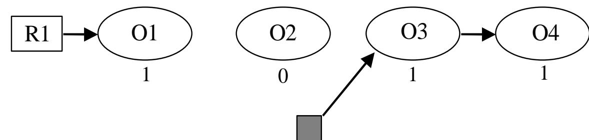
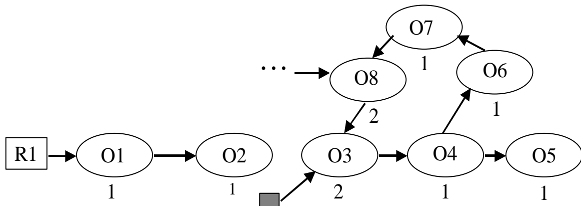


Figure 21 Reference counting is null for all unreachable objects

These unreachable objects liberate memory spaces, these at last can be directly recuperated by new objects or involve a memory space reorganisation by a reallocator. The GC will pass to the reallocator free space addresses of the same manner that the transaction passes them to the GC, the GC passes them to the reallocator.

V.5.2 Example 2



Let us graph a cycle in Figure 22.

Figure 22 Graph with a cycle in a DB

The reference counting is decreased to one. Thus, the values of reference counting is one; a research of SCC must be begun from object O3 to determine if it is reachable or not. We obtain, after deleting links between SCC's objects, a new graph (Figure 23).

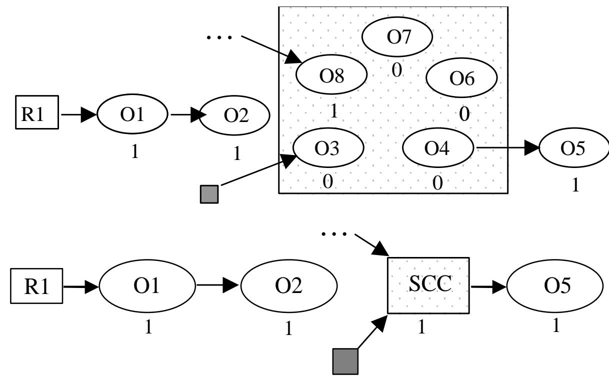


Figure 23 Detection of large cell

After reduction of the graph, the reference counting of the SCC represents the sum of reference counting of each object of the SCC. The figure 24 shows the new graph after reduction.

Figure 24 Reduction of graph

If the sum is 1 (case of configuration2 IV.3.2 in the general algorithm) then the GC cannot conclude about this branch. The GC processes another branch indicated by the transaction.

V.5.3 Example 3

Lets use an example (Figure 25) when the previous case occurs but another object is lost a incoming reference.

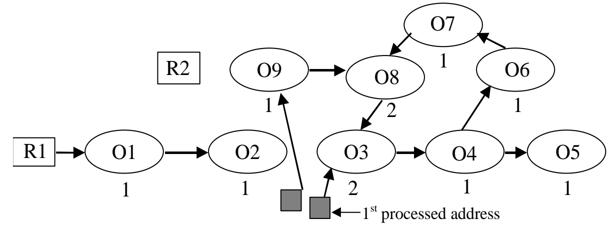


Figure 25 Addresses of two objects getting by GC

The case below (Figure 26) occurs after the same process as Figure 24.

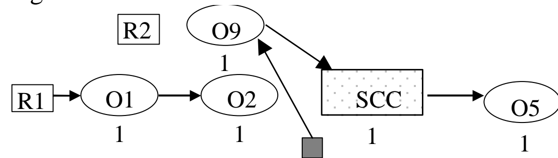


Figure 26 Detection of SCC

At this point, the GC begins again its useful treatment from object O9 (Figure 27).

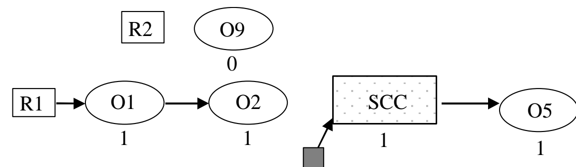


Figure 27 Processing about second object's address

The object O9 is recuperated. The GC descends on its next object that proves to be the SCC. This is destroyed since its reference counting is set to 0. Then the GC processes on object O5 and that will be destroyed also. We will end finally on the graph represented by the figure 28.

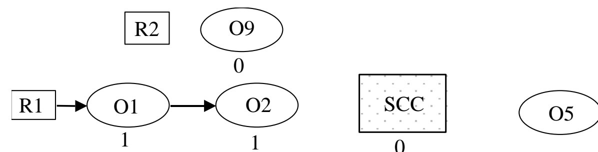


Figure 28 Graph at the end of GC processing

Finally, all objects will be found unreachable in the timely moment. The advantage of this method, is a minimal user penalisation since the process undertakes on susceptible objects to be unreachable. But, a possible penalisation can be occur during the research of SCC. However, this is minimal compared to the great advantage that this method

brings , since it allows to recuperate objects one by one (incremental) for a weak cost.
Remark: the GC could begin from object O9. The second pointer had located on object with outcoming reference counting equal to 0. Thus, we know the object has been processed previously.

VI. Conclusions and Future works

The memory management has always been a main problem during application running. At present, in the way of hardware, the relative cost of this memory has strongly decreased. On the other hand, the new data representations (images, sounds, etc.) and the communication capabilities have increased the data volume used by applications. Moreover, this data evolves very quickly. And some data becomes obsolete after some months, and even some days.
The first garbage collectors which managed memory stopped the running applications. This is unacceptable. New ways of research try to overcome this problem by using garbage collectors processing in the background of applications.

Two relevant ways are known as reference counting and mark. [JON99]. In the OODB context, they must be optimised. In fact, the reference counting is incremental and used alone but cannot detect the cycles of unreachable objects. The mark is very costly in time because of the tracing of an entire OODB. But it does not guaranteed the detection of all unreachable objects. Remember:

- the improvement of security by using locks in reading or writing in the mark and sweep; but the time cost increases!
- The improvement of the process speed to doing together the mark step and sweep step in the recopying; but the fragmentation cost increases!

We propose a incremental garbage collector (GC) concurrently to transactions. The main objective is to not stop the transactions during reclaiming and reallocating the free memory space; so, this GC processes in background.
The GC works about the unreachable objects from the objects' graph. The algorithm of the objects process seems relevant if the database has no big cycle (in term of objects quantity). But, this algorithm must be improved if the database has big cycles. Deleted objects set memory space which is recovered by objects reallocator.
The future works will concern the improvement of algorithm and of reachable objects regrouping in the same page by the reallocator. We suggest several solutions to integrate the running of the reallocator in the global management of memory. The first one consists to run the reallocator by GC after the end of the deletion of unreachable objects. This solution allows recovery of most pages during transactions. The second solution consists to run the reallocator after database saving.
The choice of a solution depends on performance tests of efficiency. We shall also try to test the performances of parallel GC and with multiprocessors. Then, we shall improve the GC implementation to use the journal of databases to know exactly the nature of cycles (size, quantity of objects, etc.).

Acknowledgments

We thank Mark Hogarth for his help with translation.

References

1. Jones R, Lins R. "Garbage Collection – Algorithms for Automatic Dynamic Memory Management". John Wiley and sons Editor 1999
2. Cooper R. "Object Databases – An ODMG approach", Vol 1. *Database technology series*. Thomson Computer Press, 1997.
3. Collins G. E. "A method of overlapping and erasure of lists". *Communications of the ACM*. 1960; 3(12):655-657
4. McCarthy J. "Recursive functions of symbolic expressions and their computation by machine". *Communications of the ACM*; 1960; 3:184-195
5. Minsky M. L."A Lisp garbage collector algorithm using serial secondary storage". In: *Technical Report Memo 58(rev.)*. Project Mac, MIT, Cambridge, MA, 1963
6. Amsaleg L. *Conception et réalisation d'un glaneur de cellules adapté aux SGBDOO client-serveur*. PhD thesis, University of Paris IV, Paris, 1995
7. Mulatero F, Thevenin J-M, Bazex P. "A global Garbage collector for Federated Database Management Systems", In: Wagner R. R. (ed) *Database and Expert Systems Applications*, IEEE Computer Society, Wien, 1998
8. Bergé C. "Graphs and Hypergraphs", North Holland Publ. Comp., Amsterdam, 1973
9. Froidevaux C, Gaudel M-C, Soria M. "Type de données et Algorithmes", *Collection informatique*. Ediscience International, 1993
10. Gondran M, Minoux M. "Graphes and Algorithmes", *Collection de la direction des études et recherches d'électricité de France*. Eyrolles, Paris, 1990