ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Comparative Study: Garbage Collector in OODBMS

Abhay Kumar [1], Jitendra Singh Yadav [2]
[1,2] Computer Science And Engineering, JECRC UNIVERSITY, India
abhaykumar.it@jecrc.ac.in

Abstract

This paper addresses the use of garbage collectors for efficient garbage collection in a large object-oriented database. The OODB is partitioned and grouped independently by using information about inter-partition references. This maintains the information on disk so that it can be recovered after any kind of crash like disk failure. We have discussed the part of garbage collector responsibility for maintaining information about inter-partition references and how they work during the transaction call. This paper also contains the comparison between the garbage collector to identify the problem in maintaining the transaction call for large dataset and a proposed solution so that uninterrupted process can be made.

Keywords: OODBMS, inter-partition references, Garbage Collector, Transaction, Partition

Introductions

Object Oriented Database System made a successful path for any database technology to take part in the field of ecommerce or we can say that due to OODBMS today the transaction of data are successful in the current contrast for example transaction of money using an Automated Teller Machine(ATM) or ordering of products through online shopping sites.[1]

Two concepts were developed using different object oriented languages (java) that is now used in OODBMS which directly supports a complex, interconnected data and an idea of object identity separated from object value.

The two concepts are: [2]

- The retrieval of storage for continual
- Memory object that are no longer accessible

From different research article it is been summarized that the inaccessible data doesn't affect the functional behavior of a running application but it shows a bad impact on its performance, as these unreferenced data increases the effective size of database and can increase access time. Automatic garbage collection is widely recognized as a fundamental mechanism that relieves software programmer from dealing with memory de-allocation. The garbage collector is designed to organize and update the information avoiding disk accesses and dangling references. These problems are present in old programming language like c which doesn't support object oriented concepts. [2]Benefits of garbage collector usage are:

- It detects all the self-referential data object of garbage like basic reference counting.
- It allows transactions to run concurrently.
- It helps in data recovery during system crash.

From the above now we can conclude that-The term "garbage collection" is usually used to refer collectively to all techniques for automatic memory management, and therefore, reference counting can be thought of as a form of garbage collection.

Related work

Object oriented paradigms forms a new form of database called object oriented database. This database made large and bulk Transactions possible over the Internet. As OODB used in transaction process the system handles the processing of data in a parallel manner but sometimes system has to suffer from shortage of memory. So to deal with this problem garbage collector concept was used. This idea comes from various object oriented languages which being used now days like java. If we do not use this concept then due to lack of storage the running transaction may be affected and suddenly stops. [3]Garbage collector also used for several issues:

- **Disk-resident data**-the size of object data can be very large and only part of database can be cached into main memory the garbage collector has to minimize the number of disk I/O's and must also avoid replacing recently fetched object in the cache.
- **Fault Tolerence**-the collector has to preserve the transaction semantics. It must survive the system crashes and must not leave the database in the inconsistent state. The recovery process must

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

remain fast even in the presence of garbage collector.

- **Concurrency**- the collection process must be able to run concurrently with client transaction. It should not affect the performance of the client operations.

As we know day by day transaction are increasing exponentially so to control the load of the transaction over the internet clustering of data is done for the fast retrieval of result.

[3]Garbage collector scheme has several advantages:

- It is possible to make the partition size small enough that the entire garbage collector can performed in main memory. This makes GC fast and reduces the number of I/O operation executed by the GC.
- This is scalable because the work carried out by the GC is independent of the database.
- The collector is free to select which partition to collect.
- Collector can run concurrently with the client activities.
- The scheme is fault tolerance.

After going through most of the research article it come to be known that the biggest problem with reference counting is its inability to handle self-referential data structures.

Garbage Collector

[2]Automatic memory management, or garbage collection, provides significantly benefits to software engineering. Automatically reclaiming unneeded data prevents memory leaks problem from unreachable objects known as "dangling pointers" (when a programmer accesses previously freed memory), and security violations. Garbage collection also improves software modularity by eliminating object ownership and reclamation problems that arise when memory passes across each module boundaries. Because of these programmers are increasingly using garbage collected languages such as Java and C#.

There are two basic Garbage Collectors used; they are Copying Collector based and Mark-Sweep based.

1. *Copying Collector based*: The copying collector algorithm re-clusters objects dynamically; the re-clustering can improve locality of reference in some cases, but may destroy programmer specified clustering resulting in worse performance in other cases.
2. *Mark and Sweep based*: the Mark and Sweep algorithm marks all live objects by traversing the object graph and then traverses (sweeps) the entire database and deletes all objects that are unmarked.

With both the above algorithms, we conclude that cost of traversing the entire object can be probably expensive for databases larger than the memory size, particularly if there are many cross-page references.

Comparison

[3]<u>Mark and Sweep based:</u>

When we use mark-sweep technique, unreferenced objects are not reclaimed immediately. The process starts after all available memory been exhausted. When it happens, then execution of the program is suspended temporarily while the mark-sweep collects all the garbage. Once the unreferenced objects have been reclaimed, the normal execution of the program resumes.

The mark-sweep algorithm is also called a tracing garbage collector because is traces out the entire objects that are directly or indirectly accessible by the system. The objects that a system can access directly are those objects which are referenced by local variables on the processor as well as by any static variables that refer to objects. These variables are called the root of an object in context of garbage collection. An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object which is also said to be live. Conversely, an object which is not referenced (directly or indirectly) is garbage.

The mark-sweep algorithm consists of two phases:

- In the first one, it finds and marks all live objects. The first one is called the mark phase.
- In the second, the garbage collection algorithm scans through the parse tree and reclaims all the unmarked objects. The second is called the sweep phase and the algorithm can be expressed as followed in fig:

Figure

```
for each root variable r
   mark (r);
sweep ();
```

[3]*Fig: Mark-Sweep Process*

In order to distinguish the live objects from garbage one, we record the state of an object, i.e., we add a special Boolean field to each object called marked. All objects are unmarked by default when they are created. Thus, the marked field is initially maintained false.

An object p and all the other objects which are indirectly accessible from p can be marked by using the following recursive mark method:

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

Figure:

```
void mark (Object p)
 if (!p.marked)
   p.marked = true;
    for each Object q referenced by p
      mark (q);
```

[3]*Fig: Recursive Mark Method*

Notice that this recursive mark algorithm does nothing when it finds an object that has already been marked. As a result, the algorithm is guaranteed to terminate. And it terminates when all accessible objects is been marked.
In its second phase, the mark-sweep algorithm scans throughout the objects in the heap, in order to locate the unmarked objects. The storage allocated with unmarked objects is reclaimed during each scan. At the same time, marked field of live object is set back to Boolean value false for the next invocation of the mark-sweep garbage collection algorithm:

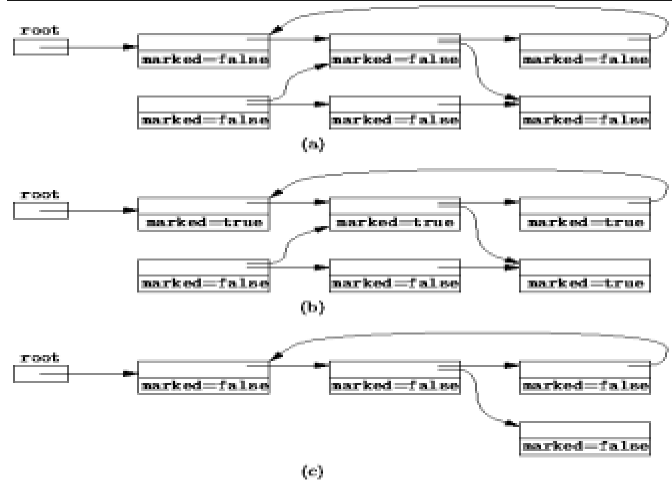
Figure:

```
Void sweep ()
 For each Object p in the heap
if (p.marked)
          p.marked = false
        else
heap.release (p);
```

[3]*Fig: The next invocation of the mark-sweep garbage collection*

Figure illustrates the operation of the mark-sweep garbage collection algorithm. Figure (a) shows the conditions before garbage collection begins. In this, there is a root variable. Figure (b) shows the effect of the mark phase of the algorithm. At this point, all live objects marked with Boolean value true. Finally, Figure (c) shows the objects left after sweep phase been completed. Only live objects remains in the memory and the all marked field have been again set to be false.

Figure:



[3]*Fig 2.2: Mark-and-sweep garbage collection.*

Because the mark-sweep garbage collection algorithm traces out the set of all objects accessible from the roots so we can say that this algorithm is able to correctly identify and collect garbage even in the presence of reference cycles. This is the main benefit of mark-sweep over the reference counting technique presented in the preceding section. A secondary benefit of the mark-sweep approach is that the normal manipulations of reference variables incur no overhead.

<u>Copying Collector based:</u>

At an abstract level, all a copying collector does is start from a set of roots and traverse all of the reachable allocated objects and then it starts copying them from one half of memory into the other. The area of memory that we copy from is called **old space** and the area of memory that we copy to is called **new space**. When we copy the reachable data then we pack it so that it continues be in a regular chunk. So, in effect, we compress the holes in memory that the garbage data taken. After the copy and compress, we end up with a compacted copy of the data in new space data and a large, contiguous memory location in new space in which we can easily and quickly allocate new objects. The next time when we again perform garbage collection, the roles of old and new spaces are reversed.
For example, let this to be a memory, where the filled boxes represent different objects and the thin black line in the middle represents the half-way point in memory.

Figure:

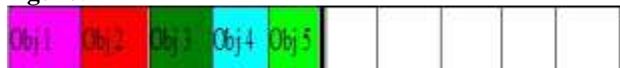


Fig: Half way point in memory [4]

At this point, we've filled up half of memory and so we initiate a collection with having old space in the

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

left and new space on the right. Suppose further only the red and light-blue boxes (objects 2 and 4) are reachable from the stack and after copying and packed in, we would have a picture like this:

Figure:

Fig: Object beyond Half way point in memory [4]

Notice that we copied the live data (the red and light-blue objects) into new space and live the unreachable data in the first half so that we can now "throw away" the first half of memory (this doesn't really require any work):

Figure:

Fig: Memory free the unreachable object [4]

After copying the data into new space, we restart the computation from where it was been left off. The computations continue allocating objects, but this time it allocates them in the other half of memory (i.e., new space). The fact that we compact the data makes easy for the interpreter to allocate the entire live object, because it has a large, contiguous lump of free memory. So, for an instance, we might be able to allocate few more objects:

Figure:

Fig: Adding new object in memory [4]

When the new space fills up and we are ready to do another collection, we flip our concept of new and old. Now old one is on the right and new one on the left. Suppose now that the light-blue (Obj 4), yellow (Obj 6), and grey (Obj 8) boxes are the reachable live objects then we copy them into the other half, throwing away the old:

Figure:

Fig: Memory free the unreachable object [4]

Conclusion

Mark and Sweep based:

Disadvantage:

The mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs so this can be a problem in a program that interacts with a human user or that must satisfy real-time execution. For example, an application that uses mark-and-sweep garbage collection becomes unresponsive periodically.

Solution:

So to reduce this cost an alternative algorithm steps were taken in which the database is divided into partitions consisting of a few pages. Each partition stores inter-partition references which references to objects in other partitions, in a persistent data structure. Objects referred from other partitions are treated as if they are reachable from the persistent root, and are not garbage collected even if they are not referred to from within the partition. Thus, partitioning makes the traversal more efficient.

Copying Collector based:

Disadvantage:

What would happen if we perform a copying process but there's no extra memory hole left over? Typically, the garbage collector will ask the Operating-system for more memory space and if the OS says that there's no more available, then the collector heave up its hands and terminates the whole program.

Solution:

The solution can be merging the idea of both Garbage Collector and can be called a **Hybrid Collector Approach**.

Benefit:

We know that there are going to be a lot of objects that are created during program initialization and which persist for the entire duration of the program. The compiler knows which objects these are and can shunt them to a separate area of memory that isn't subject to garbage collection. You can also reduce the impact of large objects by moving them off into a special memory space of their own. The so-called "large object space" would be managed separately by mark-sweep garbage collection and also these two techniques can set a long way towards reducing the position of the spaces controlled by copying collection.

Future work

We have observed that just after creation of the datasets, garbage collection has to perform extra work to convert weak pointers into strong pointers. However, once the conversion has been performed, a good set of strong pointers is established, and the further cost of garbage collection is quite low. We can develop bulk-loading techniques for reducing the cost of setting up pointer strengths.

References

1. *Bruno R. Preiss, B.A.Sc., M.A.Sc., Ph.D., P.Eng., Associate Professor, Department of Electrical and Computer Engineering,*

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

University of Waterloo, Waterloo, Canada, "Data Structures and Algorithms with Object-Oriented Design Patterns in Java".

2. *MATTHEW HERTZ, Graduate School of the University of Massachusetts Amherst, Graduate School of the University of Massachusetts Amherst," QUANTIFYING AND IMPROVING THE PERFORMANCE OF GARBAGE COLLECTION", partially submitted thesis in September 2006.*
3. *http://www.cs.cornell.edu/courses/cs3110/2014sp/lectures/26/memory.html http://www.cs.canisius.edu/~hertzm/thesispdf*