Concurrent Garbage Collection in O_2

Marcin Skubiszewski[†*]

[†]INRIA, Rocquencourt
78153 Le Chesnay, France
FirstName.LastName@inria.fr

Patrick Valduriez[†]

[*]O_2 Technology
7, rue du Parc de Clagny
78000 Versailles, France
http://www.o2tech.fr

Abstract

We describe a concurrent garbage collector (GC) for object-oriented databases. Our GC uses a new *synchronization mechanism* (mechanism that allows the GC to operate concurrently with ordinary users of the database), called *GC-consistent cuts*. A GC-consistent cut is a set of virtual copies of database pages. The copies are taken at times such that an object may appear as garbage in the cut only if it is garbage in the system. Our GC examines the copies, instead of the real database, in order to determine which objects are garbage.

GC-consistent cuts are easy to implement by already-existing code that implements consistent read-only transactions. Our GC scales up. Unlike other scalable GCs, it does not require the user to explicitely partition the database into loosely-connected subsets, and does not introduce code that must run all the time, hereby avoiding to slow down the system while the GC is not running.

1 Introduction

Automatic garbage collection is widely recognized as a fundamental mechanism that relieves software developers from dealing with memory deallocation. Unfortunately, garbage collectors (GCs) tend to be highly obtrusive, and to impose inconvenient synchronization requirements upon the rest of the system. Unsophisticated GCs block all other activities in the system, because they are based on the simplistic idea that the correctness of the GC depends on the fact that nothing is modified in the system while the GC is examining it in order to determine which objects are garbage.

More sophisticated GCs can execute concurrently with the applications. This implies that objects in the system may be modified at any time while the GC is examining them, and that the GC must implement a *synchronization mechanism*—a mechanism that ensures the correctness of garbage collection in spite of the modifications.

The synchronisation mechanism that is both the oldest known and the most widespread today is called *write barriers*. It was introduced by Dijkstra *et al.* [5]. See Jones and Lins [9] or Wilson [14] for a complete description of the state of the art concerning write barriers.

When a write barrier is used, the GC makes no effort to obtain a consistent view of the system (each object is seen in the state in which it happens to be when the GC looks at it). Instead, the system notifies the GC of every pointer modification performed while the GC is running. For this purpose, user code is instrumented, or virtual memory mechanisms are used to detect writes, or, in systems with logging, the log is made available to the GC and analyzed by it. The notifications are used by the GC to build a list of objects that were reachable at some point during the garbage detection process, yet that risk being improperly seen as unreachable. The GC considers all objects in the list as reachable; this is sufficient to ensure correctness.

In this paper, we focus on concurrent garbage collection in object-oriented databases. When used in this context, the existing garbage collection methods exhibit three problems. First, they are complicated to implement, and interact with the rest of the system in a nonmodular way. The complexity is due to a large extent to the fact that a GC-specific synchronization mechanism is needed, in addition to the standard mechanisms that already exist in the DBMS. The lack of modularity results from the fact that the GC, and particularly its synchronization mechanism, depends on many implementation details of the underlying system.

Second, the code that implements existing GCs degrades the performance of the system, by its mere existence: GC-specific tests are performed and GC-specific information is gathered all the time, even while the GC does not run. For example, Amsaleg *et al.*[1] have measured the over-

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

head caused by their GC to be between 0.6 % and 5.8 %, depending on situations.

Third, the existing GCs do not scale up in the context of databases. A solution to this problem consists in dividing the storage into *partitions*, and in collecting the partitions separately [1, 11]. But this solution has serious drawbacks. It creates a new system administration burden: objects must be placed in partitions in a way that minimizes the number of inter-partition pointers. It degrades the performance of the system: the system must maintain tables listing all inter-partition pointers, and this requires every write to a pointer variable to be accompanied with instructions that test whether the value written points to a remote partition, and that update the tables if necessary.

In order to address these problems, we have developped a new concurrent garbage collector for DBMS. This garbage collector has been implemented in O_2, a commercial object-oriented DBMS [2]. To the best of our knowledge, no other concurrent GC has ever been implemented in an industrial DBMS, although many industrial garbage collectors exist in other contexts, and are concurrent.

Our work is based on a new synchronization mechanism, named *GC-consistent cuts*. This mechanism is essential for keeping the implementation simple and modular, and for avoiding performance degradation of the whole system. GC-consistent cuts resemble the synchronisation mechanism normally used for consistent reads of a database. To implement them, it is sufficient to modify in a minor way the already-existing code that implements consistent reads. GC-consistent cuts cause no observable performance degradation while the GC is not executing.

The theoretical foundations of GC-consistent cuts can be found elsewhere [12, 13]. Work [12] contains formal proofs of all the facts about GC-consistent cuts that we quote in this paper.

Scalability is obtained in our GC by a surprisingly simple technique, that consists in ordering accesses to database pages in a way that minimizes swapping. This is efficient, because swapping is the major source of performance problems when collecting garbage in a large database.

The paper is organised as follows. In Section 2, we describe the assumptions under which our garbage collector works, and the requirements that it satisfies. These assumptions and these requirements are those of O_2; they are satisfied in most object-oriented DBMS. In Section 3, we describe the principles according to which our GC works: the synchronization mechanism (namely GC-consistent cuts) and the garbage collection algorithm. Section 4 describes the implementation of our GC, and Section 5 discusses performance results obtained with this implementation. Section 6 summarises our contribution.

2 Problem Formulation

In this section, we describe the assumptions about the DBMS that are used by our garbage collector and state the requirements that we impose upon the garbage collector.

2.1 Transactions and pages

A *database execution* is a sequence of transactions executed during a time period. Each transaction locks the data to which it has access, in either read only or read-write mode. By monitoring locks, an observer can learn which data are read or modified by any given transaction. This knowledge is an essential prerequisite for building GC-consistent cuts of a database.

We assume that transactions are atomic and serializable [7]. Serialisability means that everything happens as if the transactions were executed sequentially, in some specified order. In reality, transactions may be executed concurrently, and serialisability is implemented by the locking mechanism, which permits concurrent execution only when it is indistinguishable from a sequential one. Serialisability, and the resulting apparent lack of concurrency, allows us to depict each transaction as a fictitious atomic (thus, null-duration) event, that takes place at the time when the real transaction commits.

We assume that the database is divided into pages. For every object x, $P(x)$ denotes the page to which x belongs.

2.2 Reachability

We use a model of reachability based on the fact that before accessing an object, user code (in our case, a transaction) must first access a pointer to it. This model is commonly used in object-oriented systems, including for example O_2 and ObjectStore [10].

The database is assumed to contain a fixed set of indestructible objects called *roots*. Pointers to roots are system constants, to which all transactions have access. Moreover, every transaction has access to pointers to the objects that it has created. Outside of these two cases, objects can only be accessed by a transaction once this transaction has read a pointer to the object, from a pointer field in another object present in the database. We assume that unambiguous rules exist to determine which parts of an object are pointer fields. In O_2, for example, this is accomplished using type information present in the object's header.

There are no other possibilities for a transaction to obtain a pointer value. For example, it is illegal to perform pointer arithmetic or to store pointers in places other than pointer fields of objects.

An object is said to be *reachable* at a given time t iff it exists and the first transaction that will take place after t can access it according to the rules above. The following is a correct characterization of reachability.

Definition 1 (reachability and garbage in databases) *The* reachable *objects in a database execution E at time t form the smallest set such that* (i) *roots are reachable and, recursively,* (ii) *if at time t object x is reachable and object y exists and x contains a pointer to y, then y is reachable at time t.*

An object is garbage *at time t if it exists but is not reachable at this time.*

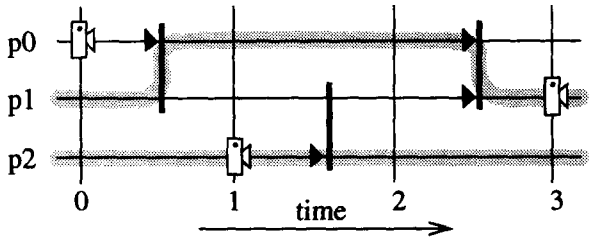


Figure 1: Example of a database execution.

We allow user code to explicitly delete objects. A destruction is considered as a special case of a write access to the object. It can therefore only take place in a transaction that has access to a pointer to the object, and has locked for writing either the object or a bigger entity (usually, the page) to which the object belongs.

2.3 Requirements Regarding the GC

Our GC is concurrent. It is required to be safe and complete. Safety means that only garbage objects are deleted. All garbage collectors must be safe. Completeness means that the GC will delete all the objects that are garbage when its execution begins. No similar guarantee is required about the objects that become garbage while the GC is already in operation (such objects are guaranteed to be deleted by the next execution of the GC). The lack of completness is a serious drawback, that we are not willing to accept. (It is sometimes accepted, but only in contexts where completeness cannot be provided [4, 3].)

2.4 Summary

The problem can be stated as follows. We consider an object-oriented DBMS, where transactions are atomic and serialisable, and where the database is divided into pages. We assume that each transaction T in the DBMS has only access to root objects, to objects created by T, and, recursively, to objects pointed to by pointer fields of objects accessed by T. Objects that exist, but cannot be accessed by future transactions because of the rules above are called *garbage*.

Under these assumptions, we want to design a garbage collector that is concurrent (works without interrupting the normal operation of the database), safe (only deletes garbage objects) and complete (deletes all the objects that are garbage when the GC starts operation).

3 The Garbage Collector—principles

In this section, we describe our garbage collector. First, we introduce the notation. Then, we introduce GC-consistent cuts, which our GC uses as its synchronization mechanism. Finally, we describe the GC itself and the algorithm that builds GC-consistent cuts.

3.1 Graphical Notation and Transaction Clock

We graphically represent database executions as follows (see Figure 1). Time flows from left to right. Each page is represented by a thin horizontal line. Each transaction is considered as a null-duration event and represented by a thick black vertical line. If a transaction reads a page, the corresponding lines cross; if it also writes the page, an arrow is drawn at the crossing. For example, the leftmost transaction on the figure reads and writes page 0, reads page 1, and does not access page 2.

When talking about a database execution, we use a special real-valued global clock called *transaction clock*. This clock takes value 0 at some time before the first transaction. Then, in an execution including n transactions, it takes each integer value $t \in [1\,..\,n-1]$ at some time between the t-th and the $t+1$-th transaction. Value n is taken at some time after the n-th transaction. For every t, during the t-th transaction the value of the clock is strictly included between $t-1$ and t. (We do not assume that the DBMS has access to the transaction clock; we only use the clock as a theoretical tool for talking about the DBMS.)

Two elements in Figure 1, namely cameras and very thick gray lines, are explained below.

3.2 Cuts

When the contents of page i at time t is recorded for the needs of the GC, the recording is called a *snapshot* and noted (i,t). Since we consider transactions as atomic events, we only take into account the possibility of taking snapshots between transactions, *i.e.* at integer times: for (i,t) to be a snapshot, t must be an integer. A *cut* is a collection of snapshots taken during an execution, containing one and only one snapshot of each page.[1] For example, Figure 1 shows a cut composed of three snapshots (represented by cameras), namely $(0,0)$, $(2,1)$ and $(1,3)$.

We say that an event happens *during* a cut C iff it happens between the times when the first snapshot and the last snapshot in C are taken, inclusively.

In order to verify the reachability of objects in a cut, we proceed exactly as if the cut was a current state of the system at some time t. In other words, to define reachability in a cut, we substitute in Definition 1 the words "at time t" with words "in cut C." This leads to the following definition.

Definition 2 (presence, reachability and garbage in cuts) *Let C be a cut. An object x is* present *in C iff C contains a copy of x,* i.e. *if the snapshot of $P(x)$ in C is taken when x exists; otherwise, x is* absent *from C.*

Objects reachable in C form the smallest set such that (i) *roots are reachable in C and, recursively,* (ii) *if object x is reachable in C and a copy of x present in C contains a*

[1] Elsewhere [12, 13] we define cuts in a more general way: a cut may contain more than one snapshot of the same page. This generalization is not useful to describe our garbage collector.

pointer to object y and y is present in C, then y is reachable in C.

An object x is garbage *in cut C iff it is present in C and is not reachable in C.*

3.3 GC-consistent Cuts

We define now two properties that a cut must satisfy in order to be used by a safe and complete garbage collector; Section 3.4 explains why these properties are important. Then, we define GC-consistent cuts, a category of cuts that satisfy the two properties.

Definition 3 (cuts exhibiting all garbage) *A cut C of a database execution E* exhibits all garbage *iff every object that is constantly garbage in E during C, is garbage in C.*

This property is satisfied by all cuts.

Definition 4 (cuts containing no false garbage) *A cut C of database execution E* contains no false garbage *iff every object that is never garbage in E during C, is not garbage in C.*

This property is not satisfied by all cuts. Figure 2 shows a counterexample: the object X is constantly reachable in an execution while a cut is being taken, and is garbage in the cut.

Before defining GC-consistent cuts, we must define *paths.*

Definition 5 (path) *Let E be a database execution, consisting of n transactions; we assume that the database contains m pages. A* path *in E is a function H that goes from the set of integer times of the transaction clock to the set of pages (in symbols: $H : \{0, ..., n\} \rightarrow \{0, ..., m-1\}$) and that satisfies, for every $t > 0$ belonging to its domain, one of the following conditions:*

1. $H(t) = H(t-1)$
2. *or the transaction that takes place between times $t-1$ and t holds locks that allow it to read page $H(t-1)$, and to write page $H(t)$.*

A path represents the way in which a pointer present at the end of a database execution E in some page i may have been successively copied during E in order to reach this page. According to Definition 5, $H(t-1)$ and $H(t)$ either are equal (this corresponds to the situation where a pointer value is not copied) or are chosen so that the transaction that takes place between times $t-1$ and t has the possibility to copy a pointer from page $H(t-1)$ to page $H(t)$. The latter means that the transaction is allowed to read page $H(t-1)$, and to write page $H(t)$.

In Figure 1, two example paths are represented by very thick gray lines (other paths exist in this execution, we just chose to represent these two as examples). The lower one is straight. This corresponds to a constant path—a path that stays in the same page during the whole execution. The upper one shows that a pointer value located in page 1 at time 3 might be there because between times 2 and 3 it was copied there from page 0, after being copied from page 1 to page 0 between times 0 and 1.

Definition 6 (GC-consistent cut) *Let E be a database execution. A cut C of E is* GC-consistent *iff it crosses every path,* i.e. *iff for each path H in E there exists some time t satisfying $(H(t), t) \in C$.*

GC-consistent cuts contain no false garbage. This fact implies that an anomaly similar to the one in Figure 2 cannot happen with a GC-consistent cut.

3.4 The Garbage Collection Algorithm

Our garbage collector is based on a classical method called *mark and sweep.* This method consists in dividing the work of the GC into two clearly distinct phases, respectively called *marking* and *sweeping.* While marking, the GC determines which objects are reachable. For this purpose, all the reachable objects are examined, according to the rules in Definition 1: roots are declared reachable; recursively, the objects that are pointed from within reachable objects are declared reachable; all other objects are considered as garbage. The marking phase of our GC only reads the database, and does not write it. (In other GCs, the marking phase writes objects: a special bit (the mark) is set in the objects that are found to be reachable. In the context of databases, however, it is more efficient to keep a separate list of reachable objects, stored outside of the database.)

During the sweeping phase, the collector deletes the objects that have been classified as garbage during the marking phase. Reachable objects are left intact.

If the GC is concurrent, a synchronization mechanism must be used during the marking phase. Otherwise, the GC may incorrectly classify reachable objects as garbage, and delete them. Amsaleg *et al.*[1] describe several example situations in which this occurs. For instance, consider the database execution shown in Figure 2, and a mark-and-sweep GC that examines the root $R1$ at time 0, and $R0$ at time 1. Under these assumptions, the GC will believe that no pointers to object X exist in $R0$ or in $R1$, and that X is garbage. X will therefore be deleted, even though, in fact, it is reachable.

The sweeping phase does not need a synchronization mechanism: here, the concurrency between the GC and the other clients of the database poses no problem, because the sweeping phase of the GC only accesses garbage objects, while the other clients only access reachable objects.

Our GC uses a GC-consistent cut as its synchronization mechanism. During the marking phase, a GC-consistent cut C of the database is built; concurrently (*i.e.* while C is being built), the GC performs marking in C, according to Definition 2. The list of garbage objects is explicitely built. Sweeping is performed once marking is finished, and consists in deleting (directly from the database, not from the

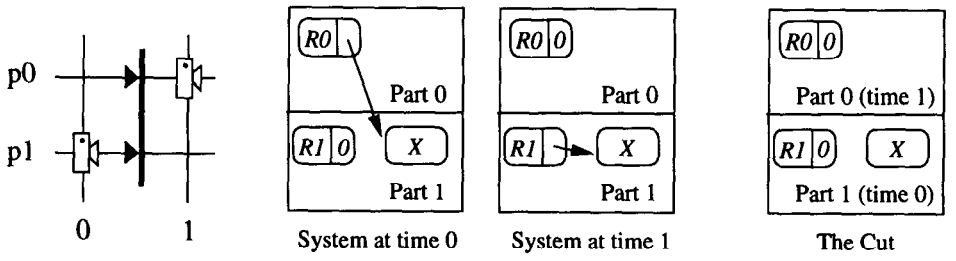


Figure 2: An object that is reachable in the execution, and garbage in a cut.

cut) the objects that were declared garbage by the marking phase.

To explain why this scheme is correct, *i.e.* why it causes the GC to be safe and complete, observe that once an object in the database becomes garbage, transactions do not have access to pointers to it, and therefore cannot make it reachable or delete it. A garbage object will therefore stay garbage until it is deleted by the GC.

When combined with the fact that GC-consitent cuts exhibit no false garbage, this observation implies that every object that is garbage in C is also garbage in the database at some time during C, and stays so until it is deleted by the GC. This, further combined with the fact that the GC will only delete objects that are garbage in C, implies that the GC is safe.

To establish completeness, it suffices to combine the observation above with the fact that cuts exhibit all garbage, and to deduce that every object that is garbage when the GC starts operation is garbage in C, and, as such, is deleted.

3.5 Building a GC-consistent Cut

Usually, the number of paths in an execution grows exponentially with the number of transactions. For this reason, it would be impracticable to directly use Definition 6 to build GC-consistent cuts. Instead, we define the notion of *captured* page.

Definition 7 (captured page) *Let C be a set of snapshots in some execution E. We say that C* captures *page i at time t iff for every path H in E such that $H(t) = i$, for some time $t' \leq t$ we have $(H(t'), t') \in C$.*

This definition means that C captures page i at time t iff C contains a snapshot, taken at time t or before, of every path H that goes through page i at time t. Intuitively, C captures page i at time t iff every pointer that is present in i at time t is recorded in some snapshot in C taken at time t or before (pointers to roots and pointers to newly created objects are excluded from this rule).

For example, in Figure 1, page 2 is trivially captured at time 1, because snapshot $(2, 1)$ belongs to the cut. Page 0 is captured at time 0 for the same reason, but is not captured at time 1 since at times 0 and 1, no snapshots are taken of the path $\{(1,0), (0,1), (0,2), (1,3)\}$, repsesented by a thick gray line.

We can characterize GC-consistent cuts as follows.

Theorem 1 (characterization of GC-consistent cuts) *A cut C is GC-consistent iff at the time when the last snapshot in C is taken, C captures all pages.*

Theorem 1 can be used as a foundation for a practical algorithm that builds GC-consistent cuts. It suffices that the algorithm ensures the following:

(a) A snapshot of every page is taken at some time.

(b) Once a page is captured, it cannot become noncaptured later.

Condition (a) implies that every page is captured at some time. This fact and condition (b) imply together that progressively, all the pages will become captured. Then, the algorithm can be stated as follows.

1. Initially, no snapshots exist.
2. If the marking process requests access to a page that does not have a snapshot, take a snapshot of this page immediately.
3. If a transaction writes a page that has a snapshot, then immediately before the transaction commits, take a snapshot of every page read or written by this transaction which does not yet have a snapshot.
4. When the marking process is terminated, halt.

A complete proof of correctness for this algorithm is provided in [12]. Here, let us just observe that rule 2 causes condition (a) to be satisfied, because the marking process requests access to all pages. Condition (b) holds thanks to rule 3, which insures that once a page has a snapshot, it will only receive information from pages that also already have snapshots.

4 The Garbage Collector—Details and Implementation

Let us recall that the objectives for our GC are simplicity, modularity and performance. Modularity means that we want the interactions between the GC and the rest of the system to be done according to well-defined rules, and to be as simple as possible. Concerning performance, our most important goals are scalability, and the absence of negative influence upon the performance of the system while the GC is not running.

4.1 Overview

Our GC is implemented by three *agents*: the *cutting agent* that builds a GC-consistent cut, the *marking agent* that performs marking, *i.e.* lists the objects that are garbage in the cut, and the *sweeping agent*, that deletes from the system the objects previously listed as garbage. The cutting agent and the marking agent run concurrently with each other. The sweeping agent runs when the two other agents have finished. All the agents run concurrently with the ordinary users of the system.

The implementation of the agents is guided by the properties of O_2. O_2 is a centralized, client-server object-oriented DBMS. A central process called *O_2 Server* manages all the data, all the logs and all the locks in the system. Other processes, called *clients*, request data and locks from the server, and perform transactions on behalf of the users.

O_2 uses *data shipping*, as opposed to *function shipping*: user code is always executed in the clients, and not directly in the server. As a consequence, the server is relatively simple. For example, it does not contain code able to create or delete objects, or to determine the type of an object, or to locate pointer fields in an object; such code is present in the clients.

Data are served by whole pages. Each page contains 4 kilobytes of data. Locks are normally granted on whole pages, but are de-escalated to single objects whenever necessary. The addresses of objects are stored on 64 bits. Each address contains a 48-bit *page ID*, representing the page where the object belongs, and a small positive integer called *slot number*. Inside each page, slot numbers are allocated consecutively, starting from 0, as objects are created in the page. The slot numbers corresponding with objects that exist at a given time do not necessarily remain consecutive, because objects may be deleted.

4.2 The Marking Agent

Principle

Marking is done according to Definition 2. Accordingly, the marking agent successively examines all the reachable objects. In order to remember which objects have already been examined, and which ones still need to be, the agent uses *three color marking*, a method introduced by Dijkstra *et al.* [5]. According to Dijkstra, at any given time an object has one of three colors. The colors have the following semantics.

black The object is known to be reachable and has already been examined.

gray The object is known to be reachable, but has not been examined yet.

white The object is not known to be reachable.

With this semantics, when marking begins, the roots are gray and all other objects are white. Marking consists in repeating the following operation as many times as possible, *i.e.* as long as there are gray objects left.

1. Select a gray object x.
2. Examine x, and color gray all the white objects that are pointed by pointer fields in x.
3. Color x black.

When marking is over, all the reachable objects have been detected as such and examined, and are therefore black. Garbage objects are white.

Details

The marking agent is implemented as an O_2 client process. It accesses the database through an application programmer interface similar to the one used by ordinary clients.

The fact that marking is done in a cut, rather than in the real database, is invisible to the marking agent: the agent requests access to objects in an ordinary manner, and the server, knowing that the requests come from the marking agent, responds by serving objects from the cut, rather than from the database.

To remember which objects are black or, respectively, gray, the agent uses hashtables hashed on page IDs. For each page ID, the table stores a bitmap that tells which objects in the page are black (respectively, gray). These bitmaps are easy to manage and small, because objects inside a page are identified by small and usually consecutive integers. There is no similar hashtable for white objects: objects that are not black or gray, are white.

The goal of the marking agent is to build a list of garbage objects. For this purpose, having a list of reachable objects is not sufficient: the marking agent also needs to know which objects are present in the cut (remember that an object is garbage in the cut iff it is present in the cut, and not reachable). For this purpose, the agent uses the fact that in O_2, the header of every page contains the list of slots used in the page, *i.e.* of slots that actually contain objects.

For every page p, the agent retrieves from the header of p the number m_p, representing the highest slot number used in this page, and the slot numbers in the interval $[0..m_p]$ that are unused (because the corresponding objects have been deleted). m_p is memorised in a hashtable. The unused slots are marked black, *i.e.* are treated as if they contained reachable and already-examined objects. This may seem surprising, but is correct, because the marking agent only needs to know two things about each slot: whether the slot contains an object that needs to be examined, and whether the slot contains a garbage object. And the answers to these questions are the same for an empty slot and for a slot containing a reachable object that has already been examined: in both cases, the slot does not contain a garbage object or an object that needs to be examined.

To summarise, the marking agent examines every reachable object in the database in order to find inside pointers

to other objects, and it examines the header of every page to find which slots in the page contain objects.

These operations may be performed in any order. We choose to order them so as to reduce swapping. For this purpose, we follow two rules. First, the agent tries to perform all the work concerning a given page at the same time. Second, whenever possible, the agent elects to perform work on pages present in the client's cache; non-cache-resident pages are brought to the client only when there is currently no work to be done on pages in cache, *i.e.* when no gray objects are present in the cache and when all the pages in the cache have already had their headers examined.

4.3 The Cutting Agent

Principle

The cutting agent can build several cuts simultaneously. Besides GC-consistent cuts, it implements *atomic cuts* and *causal cuts*. An atomic cut represents the state of the database at a given time. O_2 uses atomic cuts for consistent reads of the database. Causal cuts are an experimental feature, they are described elsewhere [12].

GC-consistent cuts are implemented according to the algorithm quoted in Section 3.5. Atomic cuts are build according to the following rule: for an atomic cut taken at time t, take snapshots of all the pages in the system at time t.

The agent implements snapshots as virtual copies: adding a snapshot to a cut consists in setting a copy-on-write flag on the corresponding page. An actual copy is made only if and when the page is subsequently modified by a transaction.

Implementation outline

We only describe the most important among the implementation choices that concern the cutting agent.

O_2 Server can run either with or without multithreading. The cutting agent is implemented as a set of C++ objects in the server; even when multithreading is used, there are no threads in the server dedicated to the agent.

A method of the cutting agent, called `notifyCommit`, is invoked before every commit. Symmetrically, method `notifyEndCommit` is invoked at the end of every commit.

While executing `notifyCommit`, the cutting agent determines which snapshots need to be taken immediately before the corresponding commit (for a GC-consistent cut, this determination is made according to the algorithm in Section 3.5). Copy-on-write flags are set accordingly. The agent then checks whether the transaction that is going to commit will modify pages that have the copy-on-write flag set. If this is the case, copies of these pages are taken, and only then the server is allowed to proceed with the commit.

In agreement with our theoretical model, the cutting agent is built upon the assumption that each transaction is an atomic and instantaneous event, that executes immediately after the corresponding call to `notifyCommit`. The agent also assumes that pages in stable storage are, at any given time, in a state that takes into account all the modifications brought by previously-committed transactions (that is, by transactions for which `notifyCommit` has been called), and no other changes.

These assumptions hold when multithreading is not used. In this case commits are indeed executed in sequence, immediately after the corresponding `notifyCommits`, and between a call to `notifyCommit` and the end of the corresponding commit, the server does nothing besides executing the commit. The locking mechanism of O_2 guarantees that everything happens as if the transactions themselves were executed in sequence, in the same order as the commits.

O_2 implements a policy called *no force, no steal* [8], according to which pages stored in stable storage faithfully represent the current state of the database.[2]

With multithreading, the situation is more complicated, because the cutting agent may read a page out of stable storage while commits are in progress. To ensure correctness despite of this form of concurrency, the agent sometimes delays taking snapshots, so as never to read a page that is in the process of being modified, *i.e.* that is written by a transaction that has already called `notifyCommit`, and has not yet called `notifyEndCommit`. Symmetrically, while a page is being read by the agent, the commits that write this page are not allowed to proceed until the reading of the page is complete.

4.4 The Sweeping Agent

The sweeping agent is simple. Its operation is driven by the hashtable of black objects, produced by the marking agent. For each page p mentionned in the hashtable, the agent deletes all the objects that were garbage in the cut, *i.e.* the objects with slot numbers less than or equal to m_p, and which are not marked as black. Destructions are grouped together into transactions.

The agent groups destructions into transactions. In the current implementation, it sweeps 100 pages per transaction. This number is not critical, but it should not be too low or too high; otherwise, either the overhead generated by transaction commits becomes significant, or, respectively, too much log space is needed.

4.5 Status of the Implementation

The implementation described here will be part of a future release of O_2. An early version of our garbage collector is

[2] Our GC can be implemented in a DBMS that does not follow the no force, no steal policy. In this case, however, constructing a snapshot is more complicated than simply copying a page out of stable storage. For example, in Exodus [6] pages in stable storage may contain changes brought by uncommitted transactions, and an undo log exists that makes it possible to suppress these changes if necessary. In Exodus, we would need to use the undo log to obtain snapshots, in addition to the pages stored in stable storage.

already available, as part of version 4.6 of O_2. This version is not concurrent, but incorporates several ideas described in this paper. Most notably, in order to minimise swapping it performs marking as described in Section 4.2.

5 Performance results

Performance measurements of the GC are currently in progress. This section provides the results that have been obtained so far.

In line with previous work (*e.g.* [1, 11]), we run several experiments that allow us to measure performance properties of the garbage collector. We attach more attention to marking than to sweeping, because marking is the hard part of garbage collection, and because in our GC only marking uses truly novel methods, that are worthy of being investigated. Our experiments measure the scalability of the GC, the ability of the marking agent to minimize swapping, and the sweeping speed (the speed at which garbage is deleted).

All the experiments run on a Sun UltraSparc 1 under Solaris 2.5. Data in our databases are distributed among three disks attached to the machine through a fast wide SCSI bus. The disks have an average seek time of 9.5 ms. The machine has 128 megabytes of RAM. We use objects that contain 160 bytes of user data. Each database page can contain up to 23 such objects (the page size is 4 kilobytes).

5.1 Preliminary Observations

We observed that when O_2 Server and the marking agent (which is executed in a client process) run on the same machine, the client process uses 35 to 50 % of the available CPU time, and O_2 Server uses less than 10 %. This observation confirms the common-sense idea that in a data-shipping DBMS, most CPU time is spent in the client. The processor is idle for at least 40 % of time because pages need to be fetched from disk, and since marking is a simple process, the time spent waiting for disk pages dominates the computing time.

When the marking agent and O_2 Server run on two identical machines, connected with a standard 10 Mbits/s Ethernet, the execution times of the GC are multiplied by the factor 3.5, as compared to the execution of both processes on the same machine. This is easy to understand, because in our experiments the disks and the CPUs are fast, and the network is relatively slow. Moreover, because the O_2 Server uses very little CPU time, putting the server and the marking agent on different machines brings practically no parallelism that could compensate for the performance loss due to the network.

The experiments described below are performed with the marking agent and the server running on the same machine.

5.2 Scalability

To assess the scalability of our GC, we ran it on databases of various sizes. Each database contains numerous objects grouped into lists of 260000 objects each. Locality is preserved: objects are listed in the order in which they are stored on disk. All objects are reachable, therefore the task of the GC consists exclusively in marking. We use a cache of 4 megabytes, but this size is noncritical due to the good locality of data.

Pointers per object	*with lists*	3	2	3/2	1
Marking time	495	717	770	818	915

Figure 4: Marking time vs. number of random pointers per object, for a database of $2^{19} - 1 = 524287$ objects, stored on 91 megabytes.

Figure 3 shows the results. We have marked databases containing up to 2.4 gigabytes of data, and at up to 12.8 million objects. The marking was done in almost-linear time, at a speed of 169 kilobytes per second or 884 objects per second.

These results are interesting, because previously-published performance reports only concern garbage collection in small databases, containing less than 128 megabytes. Our results are experimental evidence that garbage collection is practicable in real-world databases.

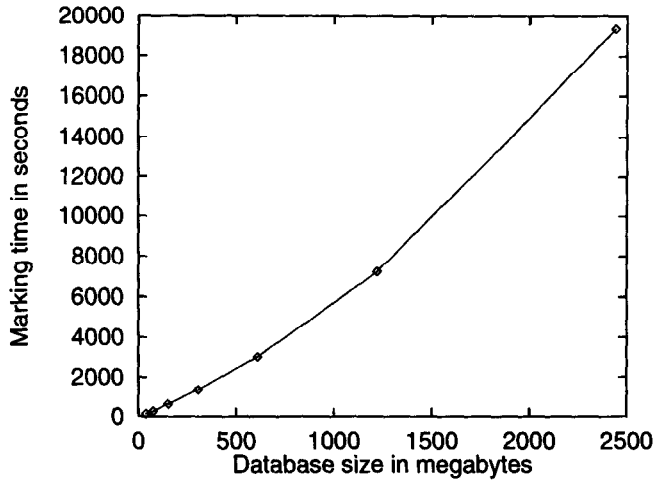
5.3 Swapping avoidance

We have measured the extent to which the marking agent manages to avoid swapping by appropriately ordering its work. For this purpose, we have built several databases with exhibit no pointer locality, *i.e.* where the fact that a given object x contains a pointer to some other object y is not statistically correlated to the way in which x and y are placed. Usually, the absence of pointer locality induces heavy swapping.

Our databases differ from each other in the number of objects that are simultaneously gray during marking. This number is important for our ordering mechanism, because work can only be performed on pages that contain gray objects. The more objects are gray simultaneously, the more likely it is that the marking agent will be able to order work so as to avoid swapping.

Our databases contain $2^{19} - 1$ small objects each, and use 91 megabytes of storage each. A cache of 2 megabytes is used. Pointers between objects are set up so that by following them, we obtain a cycle that encompasses all the objects in a pseudo-random order (in the experiments where there is more than one pointer in each object, we have several distinct cycles, corresponding with different pointer fields in the objects).

We perform four experiments, varying the number of pointers in each object. In our first experiment, each object contains one pointer. In this case, the marking agent never has more than one gray object: because there is only one root, only one object is gray when marking begins; then, the number of gray objects cannot increase: whenever the marking agent visits a gray object o, o becomes black, and at most one object (the one pointed by the pointer in o) becomes gray. Therefore, the marking agent has no



Megabytes	Objects	Marking time
38.2	0.2 *M*	2 *min* 20 *s*
76.3	0.4 *M*	4 *min* 11 *s*
152.7	0.8 *M*	10 *min* 19 *s*
305.3	1.6 *M*	22 *min* 31 *s*
610.6	3.2 *M*	50 *min* 7 *s*
1221.2	6.4 *M*	2 *h* 0 *min* 43 *s*
2442.3	12.8 *M*	5 *h* 22 *min* 52 *s*

Figure 3: Marking time vs. size of a database, for simple databases.

choice concerning the order in which objects are visited: the order is entirely determined by the pointers contained in the objects, and because the pointers exhibit no locality, the agent will swap heavily.

In the remaining three experiments, there is more than one pointer in each object: respectively, $1\frac{1}{2}$ pointers average (one object out of two contains one pointer, the remaining objects contain two pointers each), 2 pointers, and 3 pointers. Here, the number of gray object can increase while the marking agent operates: for example, when the marking agent visits a gray object o that contains two pointers, it colors o in black, but up to two other objects (those pointed to from within o) may become gray. As a result, many objects may be gray simultaneously. The higher is the average number of pointers per object, the more rapidly the number of gray objects will increase.

An extra experiment was performed under the name *with lists, with locality.* This is a mix of the experiment in Section 5.2, and of the experiment above, with three pointers per object: the objects are grouped into lists that respect locality, and also each object contains three pointers to other objects, chosen in a pseudo-random way, with no locality.

In this experiment, the marking agent can perform well, because, like in the experiment in Section 5.2, the lists allow it to access objects in the most efficient order. But because non-local pointers are present in objects, the agent might follow these pointers instead of following the lists, and perform badly. The purpose of this experiment is to confirm that this will not actually happen.

The results of our experiments are shown in Figure 4. The number labelled *with lists* represents the execution time of the experiment *with lists, with locality.* This time is equal to the marking time that results from the data in Figure 3 for an execution of the GC with lists, and without non-local pointers. This equality implies that adding extra pointers to the system is harmless for the performance of the GC, even when these pointers are nonlocal, and therefore costly to follow. The marking agent correctly choses to follow the local pointers when it has the choice between local and nonlocal pointers.

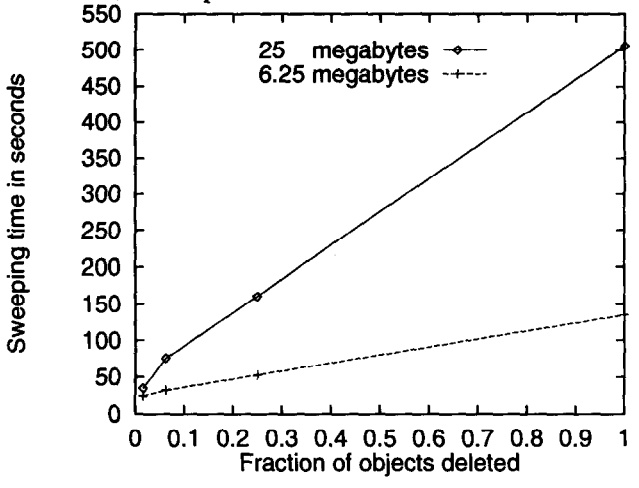


Figure 5: Sweeping times.

The other numbers represent execution times with various average numbers of pointers per object. They show that the more pointers are present (and, correlatively, the more gray objects exist simultaneously during garbage collection), the faster the marking agent works.

5.4 Concurrency

Preliminary measurements show that, as expected, the presence of the GC has no observable effect on the performance of the system while the GC does not run. When the GC executes in parallel with an ordinary client process, both the GC and the client run twice slower than usual. This is true both for the clients that only read the database and for those that both read and write many pages, and therefore force the cutting agent to take copies of these pages.

5.5 Sweeping time

We performed several sweeping experiments, removing one out of every n consecutive objects from a region of size s. We took $n = 64, 16, 4, 1$ (in the latter case, all objects

are removed), and $s = 25\,\text{MB}, 6.25\,\text{MB}$. The experiments are independent, and the database is rebuilt before each experiment.

The results are represented in Figure 5. In the figure, the x axis is labelled with $\frac{1}{n}$ (the fraction of objects removed) instead of n. The times increase steeply when n changes from 64 to 16, because for $n = 64$ only one page out of three is modified by the GC, and for $n = 16$ all the pages are (remember that there are 23 objects per page). From $n = 16$ to $n = 1$, the times increase less rapidly, because for these values all the pages in the region must be modified, regardless of n.

6 Conclusion

We have proposed a novel solution for concurrent garbage collection in object-oriented databases. The requirements for our GC are that it must be safe (only delete garbage objects), complete (delete all the objects that are garbage when the GC starts operation) and concurrent (work without interrupting the normal operation of the database). The latter implies that objects in the system may be modified at any time while the GC is examining them, and that the GC must implement a *synchronization mechanism*—a mechanism that ensures the correctness of garbage collection in spite of the modifications.

We introduce and use a new synchronization mechanism, named *GC-consistent cuts*. This mechanism simplifies the implementation, because instead of being implemented separately, it can be supported by code that exists already in many object-oriented DBMS, and that allows users to perform consistent reads of the database. Unlike other synchronisation mechanisms, GC-consistent cuts cause no observable performance degradation while the GC is not executing.

The correctness of our garbage collector has been formally established. A complete description of the theory, including proofs, is available separately [12].

Performance is improved and scalability is obtained in our GC by properly ordering accesses to database pages, in a way that minimizes swapping. This mechanism is efficient, yet much simpler than partitioning, the technique that is traditionally used to achieve scalability.

Our garbage collector has been implemented in O_2, a commercial object-oriented DBMS. To the best of our knowledge, no other concurrent GC has yet been implemented in an industrial DBMS. Performance measurements based on the O_2 implementation imply that the GC scales up, that the mechanism for ordering accesses to pages efficiently reduces swapping, and that GC-consistent cuts do not induce an excessive cost.

References

[1] Laurent Amsaleg, Michael Franklin, and Olivier Gruber. Efficient incremental garbage collection for client-server object database systems. In *Proceedings of the 21th VLDB International Conference*, Zurich, Switzerland, September 1995.

[2] François Bancilhon, Claude Delobel, and Paris Kannellakis. *Building an Object-Oriented Database: the O_2 Story*. Morgan Kaufmann, 1991.

[3] Hans-Juergen Boehm. Space efficient conservative garbage collection. *ACM SIGPLAN Notices (Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation)*, 28(6):197–206, 1993.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[5] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[6] EXODUS Project Group. EXODUS storage manager architectural overview. Available from `ftp://ftp.cs.wisc.edu/exodus/sm/doc/arch_overview.3.0.ps`, 1993.

[7] Jim Gray and Andreas Reuter. *Transaction processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.

[8] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[9] Richard Jones and Rafael Lins. *Garbage Collection*. Wiley, 1996.

[10] C. Lamb, G. Landis, J. Orenstein, and D. Weinred. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[11] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In *ACM SIGMOD International Conference on Management of Data* (to appear), volume 26, 1997.

[12] Marcin Skubiszewski and Nicolas Porteix. GC-consistent cuts of databases. Research Report 2681, INRIA, April 1996. Available from `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2681.ps.gz`.

[13] Marcin Skubiszewski and Nicolas Porteix. Partly-consistent cuts of databases. In *Proceedings of Euro-Par (to appear)*, 1997.

[14] P. Wilson. Uniprocessor Garbage Collection Techniques. In *Int. Workshop on Memory Management*, volume 637 of *LNCS*, pages 1–43, St. Malo, France, September 1992. Springer-Verlag.