Archiving Mechanism for an Object Persistent Store

Paaras Kumar (paaras@cs.brandeis.edu)

June 15, 2002

1 Abstract

This paper presents a design document of a prototypical implementation of an archiving mechanism for an object persistent database. The database on top of which this system is designed is the THOR object store[4]. The implementation of this system gives a user the ability to define 'snapshots'[1]. As this request is received by the system it assumes responsibility for automatically maintaining data from an older version without disrupting the normal services provided by the database. It uses 'Copy on Write' semantics to maintain the integrity of the older data. The system extends THOR’s 'write ahead logging' to ensure consistent recovery after a crash. It extends the MOB so that it preserves the property of write-absorption and can also support archiving. It proposes an extension to the garbage collector(GC)[5] so that new GC would run incrementally, by collecting partitions (collection of Pages) from a lower versions and moving upwards to partitions of a higher version. Finally it presents some results from some simple experiments that use OO7 benchmark traversals for evaluation.

2 Introduction

Improvements in disk storage capacities and subsequent drop in their prices have imposed new demands on designers of database systems. Among these is a desire to create a DBMS (Database management System) that can provide a facility that can store and retrieve various versions of stable data, also know as snapshots[1]. A rudimentary way of defining a snapshot would be to lock the database from further transactions, copy the entire set of objects into some backup storage and then free up the lock to let the transactions continue. This way of processing a request for a snapshot is very straightforward but its extremely inefficient. Ideally what we would like is to have a system where we can preserve the 'Snapshot Semantics'[1] but don’t have to lock the system so that it completely stops providing service to its clients.
This brings to light the notion of copy on write, where as a snapshot (or a version) is announced the DBMS acknowledges the request and prepares the system so that any further writes to the database first trigger the system to generate a copy of the old data, stores it persistently and then proceeds with further modifications. Any archiving system based on this technique must design the system so that the cost of 'lazily cloning' data will be amortized over many transactions over a period of time. Thus the 'unavoidable' overhead of cloning pages will be minimal per transaction. In this paper we propose a design that implements 'lazy cloning' and extends itself to an object database environment. Its main objectives are to provide archiving facility as transparently as possible, so that interference with concurrent transactions should be minimal. This facility should be recoverable in the presence of crashes.

2.1 THOR Model[4]

The platform used for development of this archiving mechanism is the THOR persistent object store. We assume the architecture described in Figure-1[5].

Major components of Thor’s OR (object repository) include a stable heap on the disk, a page-cache, a write-ahead-log(WAL) and, a volatile memory buffer also know as the MOB (for modified object buffer). Applications designed for
Thor access objects through a memory cache by issuing fetch request for pages.

In a normal transaction a front end will request for an object by starting navigation at some persistent root. It will request a page that contains the object. The page of an object is easily deciphered from its OREF (object reference) which is unique for each object. Once the FE is done fetching all its objects it may or may not modify them. Then, it sends a commit request along with any objects it modified or read. When the OR receives this commit request a log record is generated and it is installed in the WAL along with any pending modifications. These modifications are also installed in the MOB only after the log record has been flushed to WAL. Notice that, in Thor object shipping is used to send objects back to the OR.

Once the object is in the MOB its considered to be stable, since the entire MOB can be recovered from the WAL. However, these objects cannot stay in MOB forever. They have to be installed back to their containing pages on disk. This is done via the process of flushing log records and ensuring that the modifications associated with these log records are retrieved from the MOB and installed on disk. This process happens in the background or when the size of the log gets big.

Note that when objects are installed in the MOB, and if they happen to 'kill' a previous modification (in other words a newer copy of that object is installed) then modifications associated with a previous log entry are canceled. This results in "write-absorption". Also, note that when the previous log-entry is flushed, it will generate no disk I/O since all the modifications associated with that log entry were canceled. This has considerable impact on the archiving mechanism. This is because lazy cloning of pages is triggered by modifications applied to these pages, and if we can defer these modifications we will also push back the cloning of the affected pages. As we will see later, the cloning is deferred till the very last due to a robust MOB, which spares the ongoing transactions with any overhead archiving may incur.

Thor also has robust garbage collection process that runs at lowest priority in the background. The garbage collector (described in [5]) is responsible for collection of objects that are not referenced from any other object. This process works on individual partitions of the database. A partition is some set of pages that can easily fit in main memory. The garbage collector works by using structures called the "Translists" that are maintained for each partition on disk as objects of 'Thor universe'. These translists contain inter-partition references. For example a translist from partition P to Q will contain all the references to objects in partition Q from objects in Partition P. As objects are modified these translists have to be updated. Since these are maintained on disk, updating them synchronously after a commit is going to degrade response time. There
fore these translists are updated lazily. An in memory structure called “delta-lists” is used to capture updates to these translists. These are flushed to disk when the size of these translists grows beyond a certain threshold.

Delta-lists and the notion of partitioned collection are crucial aspects of the garbage collection. As we will see later in section 4, this allows us to build a garbage collector that work over the entire archive in a straight forward way.

2.2 Related Work

Adiba et al. in [1], which is a 20 year old paper clearly lay out the semantics of 'snapshot', but falls short of defining a efficient policy for populating content of a snapshot. Some techniques mentioned in this paper calls for explicitly copying data from a source database into backup database which will define the 'snapshot'. The logical content of the snapshot is then dependent on the user program that transfers the data. Another very similar approach which was proposed by the authors was to employ stored procedures in the source database. This procedures would then automatically be executed when a snapshot is requested and populate the snapshot. In contrast our system uses the same snapshot semantics defined in this paper but provides a transparent and a more efficient way to populate the content of a snapshot.

Santry et al. in [7]provide a file system know as “Elephant” that uses similar semantics of 'copy on write'. However, the granularity at which their system uses this technique is at a block level. There system also does not use 'lazy cloning'. Instead the blocks are copied at the first write. This may increase the response time for the users. There retention-policies were also not very strict. Instead they used heuristic based approach to retain old version by categorizing them in four policies. Keep One, Keep All, Keep Safe and Keep Landmarks. In contrast our system uses copy on write at the Page level. 'Lazy cloning’ is exploited fully and copying is deferred till the very last. Our retention policy is strict an we never loose any data.

Postgress database by Stonebraker et al.[8] use a similar compression technique to ensure that updates to data generate a new version. When a new record is installed it is written as it is and is called the 'anchor-point’. When a record is updated the new record differs from the old record in only few fields. So only the difference between the anchor point and the new record is stored. The stored record is called the delta-record and is pointed to by the anchor-point using a pointer(PTR). To retrieve the current version of record, one has to locate the anchor point and follow the delta-records linked to it using the PTR. Some notable differences between this scheme and ours are

1. The granularity at which these updates are stored. In Postgres, updates are recorded at a block level. In Contrast our system uses page level separation.
2. In our system the current version is always up-to date. Mounting older versions will require traversing meta-data and building it page-by-page. In PostgreSQL, its the other way round. To build the current version of a record you will have to traverse delta-records by starting at the anchor-record and chase PTR down all the way up till you reach the last delta-record.

SHORE [3] is a similar object persistent database as Thor. The work on Shore ended in 1997 and as far as we know there is no support for an archival system in SHORE.

Venti [6] is a very recent approach to solve the problem archiving data. Venti is not a database or a file system rather it provides basic facility that can be incorporated into a file system or a database to provide archival facility. Venti provides a block level storage manager that uses content hash of the block to determine where to store it. Since the hash function is designed to be collision resistant, this inherently provides a write-once policy. A modified block is always stored at a different address. The only issue with this facility is that it assumes that the user is always asking for continuous archival. This breaks the semantics of a 'snapshot' described in [1]. Also, since this facility is a building block it cannot be easily integrated into an existing system without a considerable engineering effort.

3 Proposed Architecture

3.1 Overview

As discussed previously the archival system uses the 'copy on write' mechanism to ensure storage of old data. The granularity of this operation is a single page (8k). That is, as modifications are installed to disk, the pages affected by them are first cloned(replicated) and installed on a separate location on disk. The choice of a single page was driven by two factors. First, it fits well with the current implementation of Thor which fetches a whole segment (four pages) and updates it with any modification that are pending in the MOB. The segment size correspond roughly in size to a disk sector. Secondly, cloning pages will ease the development of enhancing the functionality of the Garbage Collector(GC) to work on archived pages. This is due to the fact that current GC works independently on partitions (collections of pages).

In the design of this system, important considerations had to be taken to make sure that we don’t lose any benefits that the existing platform (Thor) provides by the use of MOB. As we discussed previously, MOB is crucial for short response time of the Transactions as it avoids installation reads and absorbs writes. In section 3.2 we will discuss the changes made to MOB to support the notion of snapshots, without taking away any of its benefits. The approach is to split the
MOB into buckets where each bucket is an independent MOB for a particular version of objects.

After a snapshot is announced and modifications from the MOB are propagated to the disk, it triggers cloning of those pages affected by the modifications. The cloning results in new meta-data that has to be stored permanently. The meta-data is the information about the cloned pages. It contains the version of the page and where on disk that page is stored. Storage of this meta-data is done in a structure called VPT (for Version Page Table). Since updating VPT involves disk I/O, the updates are done lazily. We will discuss this structure and any other supporting data-structures that will assist in lazy updates of VPT in section 3.3.

When modifications from the MOB are propagated to disk, we need to distinguish when a page should be cloned before installing modifications and when can we apply the new modifications without any cloning. For example if an earlier modification already resulted in an archival storage of a page, then we don’t have to clone it again. We can install the new modification right away. What we need is a bit-map like structure to inform us when a particular page is modified. If a bit for a page p is set, it will imply that an earlier modification already resulted in cloning this page. This test needs to be fast so that we can proceed quickly by installing this new object where it belongs. The data structure described in section 3.4 called HAV (for Highest archived version) serves this purpose and more. It also tells us the highest version of the page that has been archived.

In section 3.5 we summarize by using a very simple example of how these data-structures will work together to provide this archiving facility.

3.2 'Versioned' MOB

An important characteristic of the MOB is its ability to absorb writes by keeping modified objects in main-memory and allowing new modifications to replace the old ones. If we are to preserve this property of the MOB and introduce the ability to create snapshots we need to make some changes to the current implementation. MOB has to be divided so that when snapshot 'i' is announced we create a conceptual partition in the MOB so that any new modifications associated with a commit of a transaction after the snapshot 'i' is announced are installed in this new partition. We call this a 'version-bucket' and, any object that is installed into this bucket will be associated with version 'i' forever. The actual implementation will tweak the hash function that enters objects in the MOB to take an additional version number as an argument besides the usual OREF.
Once this change is established we have to make sure that Reads, Writes and Flushes from the MOB are tailored to this new change as well. Regular reads (fetches) will change so that now the MOB is consulted with its highest version-bucket first if we need to send the most up to date copy of the object. If object is not found then, we search our way down the MOB into lower version-buckets. Writes will be modified so that after installing the commit log-record (also stamped with a version number) when the modifications are entered into the MOB, they will be written into the corresponding version bucket. If there is older copy of the object in that bucket it will be written over resulting in write-absorption. Flushes from MOB will change so that when we apply the log-record of some version 'i', we only flush out objects from the MOB that are in version-bucket 'i'.

By applying these changes we preserved the wonderful properties of the MOB and also changed it to fit our new requirement. The overhead that will incur due to some extra checking of version-buckets will be very small since all this is done in main memory. Moreover the likelihood of having more than 1 version-bucket in the MOB at any given time is little.

3.3 VPT (Version Page Table)
As mentioned before, VPT is the data-structure that keeps track of any meta-data information that is generated when pages are cloned. The VPT itself is an array the size of total number of pages on disk. It is kept on disk alongside the segment-table (or page-table). Each entry in this array contains address of another array know as Vpt-Tuple-Array. Vpt-Tuple-Array is the actual container where address and the version-number of a page is kept (together we call it a tuple). By default we allocate a Vpt_Tuple_Array of size 128 tuples, out of which two tuples keep internal information. Effectively each tuple_array can hold entries for 126 versions per page. Figure-2(on previous page) describes the VPT and the associated structure. The first tuple-array will be initialized on demand, i.e initially VPT entries don’t point to anything.

Since Vpt_Tuple_Array information is maintained on disk, updates require disk I/O. As pages are cloned and installed on disk if we have to update the array right away that will requires us to do more disk I/O than needed at that moment. Instead what we do is add a new log record (Page-Cloned-Record) in WAL that keeps information about what page we just cloned and where we cloned it.

We also enter into a hash map a small entry called Delta-Cloned-Page-Entry (DCPE) for each page we clone, which has the same information as the Page-Cloned-Record. This way we can defer updating the Vpt-Tuple-Array till we flush corresponding log records. DCPE entries can be fully recovered from the log in case of a crash. As previously mentioned in the introduction, applying log records is done in background or when log is full.

Note that when modifications are applied it affects more than one page. Thus instead of creating one log record for a single page that is cloned, we create a single log record for multiple cloned pages. This emulates ’batch-logging’ to some extent.

3.4 HAV (Highest Archived Version/Bit-Map)

HAV is a simple memory structure that serves two purposes. It serves to identify whether a page has been modified or not after applying pending modifications from the MOB. Secondly it gives information about what highest version of the page is archived. For example HAV[x] returns the highest version of page x that is archived on disk. Size of HAV depends on number of pages we have on disk. Each entry is 4 bytes.

This is how HAV is used. When log records are applied and pending modifications from the MOB are flushed, we check the version of the object being flushed (remember object always belongs to some version-bucket). If the object being flushed is of version ‘i’ then we are necessarily archiving for version i-1 on disk(also know as building-version). We evaluate entry HAV[PID], where PID
Figure 3: Brief description of archiving mechanism

is the page-id of the page on which the object resides. If the value returned is less than the 'building-version', then it implies that this page is clean and has not been modified since it was last cloned. We can safely done the page and apply the modification. We then set HAV[PID] = 'building-version', so that if another modification to this page is being applied it will not result in any further cloning, until newer modifications from a higher version-bucket are being applied. Also note, that HAV[PID] is now updated to reflect the highest archived version for that page. This is useful since it will optimize the retrieval of a page when we mount an older version v. If HAV[PID] for some page is is less than v then we know that the vth version of the page is also the current version of the page. If HAV[PID] is more than or equal to v then we have to traverse VPT and the associated Vpt_Tuple_Array to find the address of the archived page on disk.

HAV is recoverable after a crash. The recovery process has to traverse Vpt_Tuple_Arrays before it can be recovered fully. If clean shutdown is requested we dump this memory structure on to disk for which space is pre-allocated at startup. During a clean startup, we first read HAV from disk and bring it back in memory

3.5 Summary

Please look to Figure 3 for reference.
Lets assume that all database assemblies are committed. MOB and the Log are empty and a snapshot ‘i’ has been announced by client . The client runs a traversal. This brings pages into his cache. He finishes the traversal by sending a commit and the objects he modified. As mentioned before object shipping is used in Thor as oppose to sending whole pages. These modified objects are flushed to Log. Logentry-1 is generated associated with version ‘i’, and then these objects are installed in MOB. In Figure-3 these are shown as objects X.1 an Y.1 and they are installed in there appropriate version-buckets. If the client happens to run another traversal and send the same but modified objects back with a second commit request then, objects X.2 and Y.2 are first flushed to log, Logentry-2 is generated and then these objects are entered into the MOB. This time around since we already had a copy of objects X an Y(in form of X.1 and Y.1) in MOB, write absorption will take place along with any cancellation of log records associated with previous entries. Recall that when X.1 and Y.1 were installed a Logentry-1 was first generated. This cancellation is done by simply decrementing a ‘pending’ variable associated with Logentry-1. In this example initial value of ‘pending’ is two (since there were two objects installed in MOB) . When X.2 is entered ‘pending’ value for Logentry-1 is decremented by 1. When Y.2 is entered into the MOB ‘pending’ value for Logentry-1 is decremented one more time. This results is ‘pending’ going to zero for Logentry-1, and the ‘pending’ value for Logentry-2 is set to two.

Now lets say that log is being applied. First Logentry-1 gets applied. We realize that the ‘pending’ value for this record is 0. This results in no objects getting flushed from the MOB. When Logentry-2 is being applied, we realize that there are two entries in the MOB that are associated with this Logentry. We search the MOB in the i-th version-bucket and find X.2 and Y.2 in the MOB, MOB is now ready to flush and potential cloning of pages can take place. Note that this is happening in the background when log is getting flushed.

Lets say object X belongs to page P and so does object Y. When X.2 is is being flushed we fetch P from disk. This is the ‘installation read’. We check HAV[P] to see if the value is less than or equal to i-I(i.e the building-version). Since this is the first time we are installing an object on Page P after a snapshot, the value must be less than the ‘building-version’. So we clone the page and set HAV[P] to i -I . This means two things.

1. The highest version installed for Page P is i -I

2. When we are about to install Y.2, we should not clone Page P since it belongs to the same page.

When the page is finally cloned we generate some meta-data associated with Page P of version i -I. A Page_Cloned_Record is generated and flushed to Log. A DCPE is generated to reflect the cloning of Page P and entered into a hash map in memory. Note that by doing this we are avoiding updating
Vpt_Tuple_Array right away. When Log is applied again this DCPE entry will be flushed out and installed in into Vpt_Tuple_Array corresponding to Page P.

4 Garbage Collection

To understand why we need Garbage Collection (GC) in this Thor based archival system lets take a look at a simple example. Consider Object X, Y, Z. Assume that object X is on Page P1 and objects Y and Z are on Page P2. Assume that Z is orphaned. That means, it is no longer being pointed from any where. However, before it gets collected lets assume that P1 and P2 are cloned (or archived) because of some modification that is installed in these two pages. Now, P1 and P2 are archived some where on disk with object Z sitting there in its orphaned state. This needs to be fixed.

One argument that immediately comes to mind is as follows. If the the archive is suppose to be read only then why do we need to fix this problem. The answer lies in the fact that if we can get rid of object Z from the archived page and move object Y to the same page as X then, we just freed some more space from our storage media. However, this argument becomes less and less important as storage becomes cheaper and cheaper.

Assuming that garbage collection is still considered a reasonable feature to be added. The following scheme is proposed. We need to run a special version of garbage collector that will work on archived pages. Following precondition needs to be satisfied before we can rely on this scheme. We need to ensure that when snapshot 't' is announced we reliably store any 'delta-lists' that are in memory at that moment. Since delta-list absorb changes to translists, and translists are basic set of information that we need to collect individual partitions, we need to ensure that they can be reconstructed for a particular version. We can achieve this reliably by making sure that we store the delta-lists as a snapshot is announced. And, since translists are themselves Thor objects any modifications to these objects will automatically be preserved under our current archival scheme. Thus, we can reliably re-construct translists for a version x and apply the updates of the stored delta-lists to bring them up to data.

Once translists are up to date we can run GC in a similar fashion as we do it in 'regular' Thor. However, we will need to run this modified GC so that we collect the lowest version first and then move to a higher versions. This will guarantee that garbage is collected from all the versions. This assumes that orphaned objects from a lower version can never be referenced again in any higher version.
<table>
<thead>
<tr>
<th>Versioning_enabled</th>
<th>Traversal Time</th>
<th>Commit Time</th>
<th>Total</th>
<th>time to clean the log</th>
<th>log-clean splits</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>384.0</td>
<td>130.3</td>
<td>514.3</td>
<td>10.0</td>
<td>(5,0,5.0)</td>
</tr>
<tr>
<td>False</td>
<td>383.5</td>
<td>129.6</td>
<td>513.1</td>
<td>1.5</td>
<td>(1,5)</td>
</tr>
</tbody>
</table>

Table 1: 1000 transactions with one 2a traversal in each transaction

5 Basic Performance Evaluation.

5.1 Test Environment

Before presenting the analysis, let's first understand the setup and the kind of test we are running. The test bed consists of a single Object Repository (OR/Thor Server) and a single Front End (FE/client). We initialize the OR to its default settings. Which is about 4MB of cache about 6MB of Logsize. A file is used to simulate disk and the size of this disk is 200 MB. The MOB takes up about 1/6 of the Cache size. The Log is designed to flush when its 90% full and will stop flushing when its at 80% mark. The server is running Red Hat Linux 6.2 and the disk is located in a local partition so that no network access is need for disk I/O. The FE also runs on the same machine with a cache size of 10 MB.

We ran OO7 benchmark[2] traversals that do both reads and write. Specifically we ran traversal 2a and 2b. We ran these traversal on both 'small' and 'medium' size databases, with slight changes in configuration. From OO7 benchmarks, a small database is a collection of assemblies that are about 4 MB in size and a medium database is about 44 MB in size.

5.2 Experiments

5.2.1 Traversal 2a on 'small' size database (Table -1)

Here the setup is default. We ran about 1000 transactions with each transaction consisting of 1 traversal. We ran this traversal with versioning enabled and without any versioning enabled. In Table-1 the second column gives the time it took FE to do the traversal. The third column gives the time it took FE to commit. The fourth column is the total of previous two. The fifth column is the total time it took to clean the log and the sixth column is an n tuple where each entry in the tuple is a single run of “clean log”. We refer to each entry as a 'split'. Adding all the 'splits' of this tuple will give the total time to clean the log, which is given in the fifth column. All times are in seconds and are rounded up to the first decimal

This traversal gives us a glimpse of what is going on in our database. Note that for both these traversal the total time to commit is same. This suggests that versioning did not interfere with the traversal. However note that, time to clean the log is 10 seconds when versioning is enabled and 1.5 seconds when its not. Also note the split up of the time to clean the log. This suggests that the
<table>
<thead>
<tr>
<th>Versioning Enabled</th>
<th>Traversal Time</th>
<th>Commit Time</th>
<th>Total</th>
<th>Time to clean the log</th>
<th>log-clean splits</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>90.2</td>
<td>425.7</td>
<td>516.0</td>
<td>392.8</td>
<td>(53.5, 63.3, 15.5,...)</td>
</tr>
<tr>
<td>False</td>
<td>91.3</td>
<td>353.7</td>
<td>445.1</td>
<td>326.8</td>
<td>(15.9, 14.5, 14.5,...)</td>
</tr>
</tbody>
</table>

Table 2: 20 transactions with one 2b traversal in each transaction.

cloning of the pages is being done at the very last in this traversal. It probably
took 5 seconds to clone the pages and modify the original segments and took
another 5 seconds to update the VPT meta-data structures. You would think
that updating the Meta-data should take less time. However, since the Meta-
data is more scattered over the disk, it makes sense that it took about the same
time as cloning of the pages.

Another reason why cloning was deferred till the very last is the fact that we
are experiencing very good cache and MOB hit rate. This makes sense since our
database is small and all the committed objects can easily fit in MOB causing
extensive write-absorption. This results in no cloning during the traversals
because no object gets flushed out of the MOB since its busy absorbing writes.

Any archival system with page based cloning will hope that cloning is done
without any interference with the actual transactions. However, this will only
happen when we have good cache and MOB hit rate at the OR. However what
will happen if we don’t have a good cache and MOB hit rate. To see some nega-
tive effects of cloning pages we decided to run traversal 2b of OOT7 benchmark
on a large database. First of all, traversal 2b is already more extensive traversal
than 2a. It modifies about 10 times more objects than 2a. Secondly we decided
to run it on a medium size database. This will result in lot more objects be-
ing committed than our default cache or MOB can handle. This should result
in MOB misses and we should see some negative interference due to cloning.
Results of this run are presented next.

5.2.2 Traversal 2b on 'medium' size database (Table -2)

Here the setup was slightly modified. It only affects the FE. We changed the
FE cache size to be about 40 MB this was primarily so that FE can load up all
the objects of the 2b traversal in its cache and commit them at the same time.
We also ran only 20 transactions.

Notice that in this run the commit time when versioning is enabled is about
20% more than when versioning is disabled. Also observe that first two splits
of the log-clean process when versioning is enabled are 53.5 and 63.3 as oppose
to 15.9 and 14.5. The rest of the 'splits' were in the range of 14 to 15 seconds
in both cases. We had about 20 total 'splits' during this traversal. Because of
space problem on this page, We could not show all the splits of the tuple.
These results indicate that the clean-log process was initiated most of the time right before a commit. This is because, previous commit size was much larger than MOB can handle at once. Therefore log had to flush before it can accept any further commits. This also illustrates what happens when you experience a bad MOB hit rate. Note that the first two 'splits' of the clean-log process when versioning is enabled are much higher than 'splits' of the clean-log process when versioning is disabled. This indicates (as expected) that MOB was not able to handle all the objects that were committed. As the log became relatively full right after the first commit it resulted in flushing of the MOB and subsequent cloning of the pages. Since cloning generates more records with meta-data, we see that the second split is also very high compare to the second split when versioning is disabled. This must mean that the second split also includes the time it took the system to update the meta-data.

Note that after the second 'split' the rest of them are similar in both cases. Recall that once the pages are cloned, we update the HAVF data-structure to indicate that no further cloning should take place if same page is getting modified. The fact that rest of the 'splits' are similar in both cases indicates that the first two rounds of clean-log pretty much cloned all the pages that are affected by this traversal. Therefore, the rest of the 'splits' in both cases result in similar time.

6 Conclusions

In this paper we laid out a design of a prototype system that employs 'lazy cloning' along with 'copy-on-write' semantics to build an archiving facility and extended it to an object persistent database (THOR). From the results of the simple experiments we can conclude the following about this system. 'Lazy' page level cloning can work fairly efficiently when it works in conjunction with a good caching policy. As we saw in our experiments when the MOB was experiencing good hit-rate the cloning overhead did not interfere with the transactions. However, when MOB hit-rate was forced to degrade in a controlled test, the overhead of cloning pages started to interfere with transaction's response time.

7 Future Work & Work in Progress.

I would like to point out that many components of this system are still in implementation phase. These include:

1. Splitting the MOB up in versional buckets as proposed in section 3.2
2. The implementation of the proposed garbage collector as proposed in section 4.
3. Entire recovery mechanism for THOR as well as the archiving facility.
Some future work may involve storing of archived pages over the network in a distributed peer-to-peer type file system. We may also experiment with various novel batching techniques for the log. This will eliminate or reduce significantly the bottle neck that is created while writing many sequential log records for example, during cloning of segments. Also, it will be interesting to see how the database will behave when you have multiple clients accessing it. This will effectively reduce the MOB size per client and create MOB misses. We predict that we will see some performance degradation similar to the one we saw when we ran traversal 2b on a medium database.

References


