A Performance Evaluation of Log-Only Temporal Object Database Systems*

Kjetil Nørvåg
Department of Computer and Information Science
Norwegian University of Science and Technology, Norway
noervaag@idi.ntnu.no

Abstract

In most current database systems, data is updated in-place. To support recovery and increase performance, write ahead logging is used. This logging defers the in-place updates, however sooner or later, the updates have to be applied to the database. This often results in non-sequential writing of lots of pages, creating a write bottleneck. To avoid this, another approach is to eliminate the database completely, and use a *log-only* approach. The log is written contiguously to the disk, in a no-overwrite way, in large blocks. The log-only approach is particularly interesting for transaction time object database systems (TODB). While previous approaches to TODBs have been page based, i.e., when an object has been modified, the whole page the object resides on has to be written back, our approach is object based. One of the objections against operating at object granularity is that the read cost will be prohibitively high. We will in this paper show that this is not necessarily true. We use analytical cost models to compare the performance of log-only and in-place update TODBs, and the analysis shows that with the workload we expect to be typical for future TODBs, the log-only approach is highly competitive with the traditional in-place update approach.

1 Introduction

As disks gets cheaper, larger amounts of data is stored in databases. Much of this data will be defined as temporal data, where the whole history of the individual objects is kept, and data is never deleted. Because the increase in disk speed is much lower than the memory and CPU speed, we have an increasing secondary memory access bottleneck. This is not a new situation, minimizing the effects of this bottleneck has been the motivation behind much of the database related research. However, the advent of very large main memory buffers, makes it necessary to revise previous work and solutions.

In most current database systems, data is updated in-place. To support recovery and increase performance, write ahead logging is used. This logging defers the in-place update, but sooner or later, the update has to be done. This often results in the writing of lots of small objects, creating a write bottleneck. To avoid this, another approach is to eliminate the database completely, and use a *log-only* approach. The log is written contiguously to the disk, in a no-overwrite way, in large blocks. This is done by writing many objects and

*IDI Technical Report 15/99, ISSN 0802-6394.

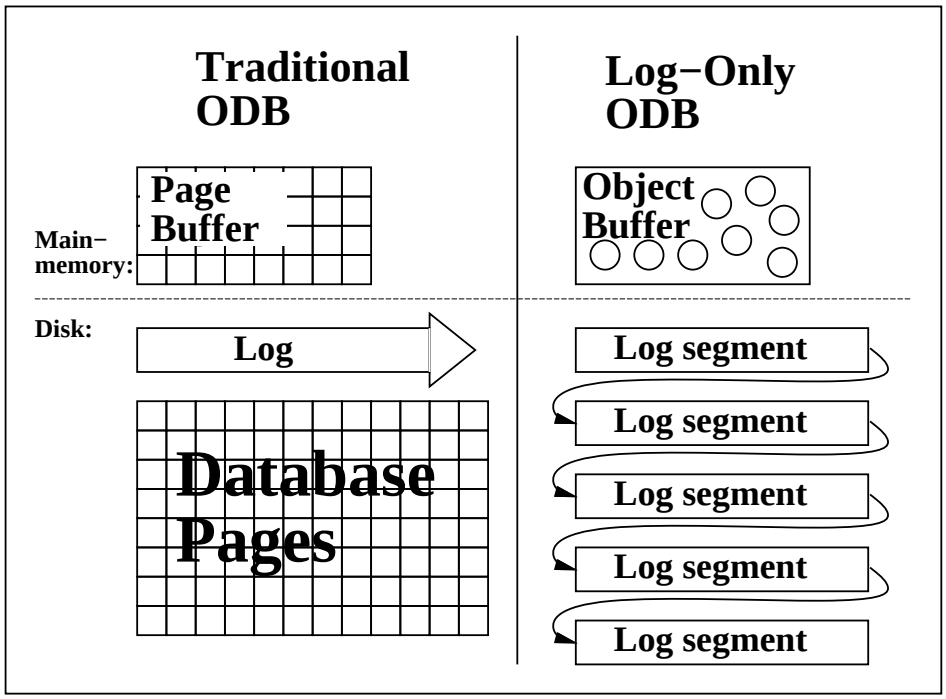

Figure 1: In-place update page server vs. log-only object database system.

index entries, possibly from many transactions, in one write operation. This gives good write performance, but possibly at the expense of read performance. Figure 1 illustrates the most important differences between a traditional ODB, and a log-only ODB.

The log-only approach is particularly interesting for transaction time object database systems (TODB). In a TODB, object updates do not make previous versions unaccessible. On the contrary, previous versions of objects can still be accessed and queried, and a system maintained timestamp (commit time of the transaction that created this version of the object) is associated with every object version. In a TODB, we want to keep previous versions of object, a feature that comes for free when the TODB is based on the log-only approach.

The log-only approach has many features that makes it interesting. This includes fast recovery and flexibility in chunk size for large objects (for example multimedia data and large multidimensional arrays, like OLAP data cubes). The log-only approach also benefits more from using RAID technology than traditional systems. The reason for this is that it writes large blocks, which is necessary to achieve high write bandwidth in RAID. Support for efficient handling of large objects is more important than ever, as more and more applications store large objects in the database systems, rather than in separate files.

Previous log-only object database systems have been page server based. While this works well in many contexts, it is not ideal. By operating on page granularity, you get many of the disadvantages of traditional pager servers. For example, if clustering is bad, and only a small part of a page has been updated, it is still necessary to write back the whole page. With bad clustering, main memory buffer utilization will be bad as well. A page based log-only ODB also makes transaction management difficult. To avoid page level locking, you essentially need to have 1) a separate log anyway, or 2) use ad-hoc techniques to solve the problem. Both

solutions are likely to hurt performance and increase complexity, and have convinced us that an object based log-only ODB is the way to go.

It is important to note that one of the main reasons why previous approaches to log-only systems have not been able to achieve significant speedup compared to traditional systems, is that except shorter recovery time, they have not tried to benefit from any of the potential benefits described above.

We are currently developing a TODB, the Vagabond[1] TODB [7, 8]. The Vagabond TODB is based on the log-only approach, and operate on object granularity. One of the objections against operating on object granularity, has been that the read cost will be prohibitively high. We will in this paper show that this is not necessarily true for a log-only TODB. We will show this by using analytical modeling to do a performance analysis of a log-only TODB (LO-TODB) versus a TODB based on traditional in-place update techniques (IPU-TODB). We will show that with the workload we expect to be typical for TODBs, the LO-TODB is highly competitive with the traditional approach.

The organization of the rest of the paper is as follows. In Section 2 we give an overview of related work. In Section 3 we give an overview of LO-TODBs. In Section 4 we outline the assumptions behind our analytical models. In Section 5 and Section 6 we present analytical models for an IPU-TODB and an LO-TODB. In Section 7 we compare the performance of IPU-TODB and LO-TODB. Finally, in Section 8, we conclude the paper.

2 Related Work

No-overwrite strategies have been used in shadow-paging recovery strategies earlier, e.g., in System R [1, 3], but with the limited buffer size at that time, the performance was not satisfactory. POSTGRES [13] also employed a no-overwrite strategy, but had also its performance problems, for several reasons, the most important being the buffer force strategy used.

Vagabond is based on the same philosophy as log-structured file systems, which was introduced by Rosenblum and Ousterhout [11]. LFS has been used as the basis for two other object managers: the Texas persistent store [12], and as a part of the Grasshopper operating system[4]. Both object stores are page based, i.e., when an object has been modified, the whole page it resides on has to be written back.

To our knowledge, there have been no publications on log-only ODBs operating on *object* granularity as in the Vagabond approach. However, the *log-structured history data access method* [5] uses some of the same ideas, The LHAM is based on the *log-structured merge-tree* (LSM-Tree) [10], which is an hierarchy of indexes. Inserts and updates are only done to the first level index, and the contents of one level in the index is asynchronously migrated to the next level. As a result, all data inserted or modified during a certain time period will be in the same level. Search for data written at a certain time is efficient, but searching for the most recent version of some data can be costly.

3 An Overview of the Log-Only Approach

With the log-only approach, already written data is never modified, new versions of the objects are just appended to the log. Logically, the log is an infinite length resource, but the

[1] From *Webster's Encyclopedic Unabridged Dictionary*: Vagabond: "a person, usually without a permanent home, who wanders from place to place; nomad". Quite similar to our objects!

physical disk size is, of course, not infinite. This problem is solved by dividing the disk into large, equal sized, physical segments. When one segment is full, writing is continued in the next available segment. As data is vacuumed, deleted or migrated to tertiary storage, old segments can be reused. Dead data, in a TODB most often old index nodes, will leave behind partially filled segments, the data in these near empty segments can be collected and moved to a new segment. This process, which is called *cleaning*, makes the old segments available for reuse. By combining cleaning with reclustering, we can get well clustered segments. In a traditional system with in-place updating, keeping old versions of objects, which is required in a transaction time temporal database system, usually means that the previous version has to be copied to a new place before update. This doubles the write cost. With the log-only approach, this is not necessary. Keeping old versions comes for free, except for the extra disk space.

Because each new version of an object is written to a new place, logical object identifiers (OIDs) are needed. When using logical OIDs, an OID index (OIDX) is needed to do the mapping from logical OID to physical location when retrieving an object. The index entries in the OIDX, the *object descriptors* (OD), contains the physical address for an object and the commit timestamp.

In a non-temporal ODB with in-place updating of objects, the OIDX needs only to be updated when objects are created, not when they are updated. In a log-only ODB, however, the OIDX needs to be updated on every object update. This might seem bad, and can indeed make it difficult to realize an efficient non-temporal ODB based on this technique. However, in the case of a *temporal* ODB, the OIDX needs to be updated on every object update also if using in-place updating, because either 1) the previous or 2) the new version must be written to a new place. Thus, when supporting temporal data management, the indexing cost is the same in these two approaches. We have in previous work developed several techniques that can be used to reduce the OIDX access cost [9, 6].

Using the log-only approach also gives new opportunities to improve performance. In order to reduce storage space and disk bandwidth, objects can be compressed before they are written. With the log-only approach, objects are written to a new location every time, so that *we only use as much disk space as the size of the current version written.* In a system employing in-place updating, it is difficult to benefit from object compression, because the compression ratio will very from version to version, and it is difficult to know how much space to reserve. Another important advantage with the log-only approach is fast crash recovery. Only one pass through the log is necessary. This is very important to achieve high availability.

It should also be noted that some of the problems in previous no-overwrite database systems have been solved in the Vagabond system. For example, algorithms for steal/no-force buffer management, fuzzy checkpointing and fast commit have been developed.

4 Analytical Modeling

Analytical modeling in database research has mostly focused on disk costs. This has been the most significant cost factor, and the CPU processing has gone in parallel with disk transfer, making the CPU cost "invisible". With increasing amounts of main memory available, this is not necessarily correct. In that context, CPU cost, and memory-to-memory transfer would be important as well. Therefore, using the analytical model presented in this paper, the results for the case when all data in the database fits in main memory should be taken with a

large grain of salt. Therefore, we only provide results for memory sizes smaller than half the database size. However, even if the actual performance would not be as high as indicated by the model, the performance would still be much higher using the log-only approach, because we in this case can use the maximum disk write bandwidth.

As mentioned previously, OIDX access costs can be high. However, in the analysis here, we limit the discussion to storage and retrieval of the objects only, under the assumptions that:

1. We have enough disks available to avoid the OIDX operations becoming a bottleneck.

2. In a traditional IPU-TODB, the OIDX will be based on in-place updating. In an LO-TODB, the OIDX can either be stored log-only, or it can be stored separately, and updated in-place. By not including the OIDX in the analysis, we assume that OIDX operations in an LO-TODB will have the same costs as in a IPU-TODB, even though it is very possible that the costs can be *less* if using the log-only approach on the OIDX as well.

For both models, we assume the total amount of buffer memory to be M. In the case of the IPU-TODB, this memory is used for buffering object pages, OIDX nodes, and object descriptors (OD cache). In the case of the LO-TODB, this memory is used for buffering objects, OIDX nodes, and object descriptors (OD cache). Because we in this analysis do not consider index performance, we will only consider the memory available for objects and object pages, which we denote as M_{obuf}.

For both systems modeled here, performance can be increased by adding more disks. Both the objects and the OIDX can be partitioned over several disks. To simplify the analysis, we assume that only two disks are available for object storage in our model. This is the smallest number of disks needed to be able to recover from media failures. For the IPU-TODB, one of the disks is used for data, and the other one is used for the log. For the LO-TODB, we use the two disks in a RAID 1 configuration (mirroring). This means that we get the write performance of one disk, but the read performance of two disks.[2]

The main purpose of the models developed in this paper, is to compare the LO-TODB and IPU-TODB approaches. Therefore, we do a conservative analysis, always trying to err on the right side. In practice, this means that we make a very optimistic analysis for the IPU-TODB, and similarly pessimistic analysis for the LO-TODB. For example, compression as described previously, is not included in the analytical model. Also, we restrict this analysis to objects smaller than one disk page. Larger objects can be written and read more efficient with the log-only approach, because of larger flexibility with respect to object sizes and sizes of subobjects in large objects. In general, the performance of an LO-TODB relative to an IPU-TODB will increase with increasing object sizes.

4.1 Disk Model.

We use a traditional disk model, where the cost of reading a block from disk is the sum of the start up cost T_{start} and the transfer cost T_{transfer}. In our model, the average start up cost is fixed, and is set equivalent to t_r, the time it takes to do one disk revolution. The transfer cost

[2]By adding more disks and using these in a RAID configuration with one parity disk, the performance improvement would be higher in an LO-TODB than an IPU-TODB, because only the LO-TODB would be able to efficiently exploit the write bandwidth of a RAID system.

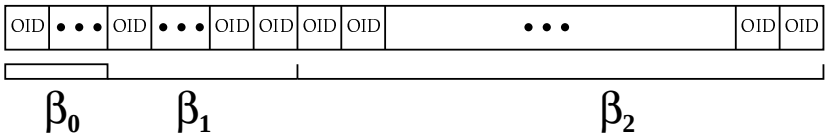

Figure 2: Partitioned data area. Each partition contains a fraction β_i of the data granules, and α_i of the accesses are done to each partition.

is directly proportional to the block size, and is equivalent to reading disk tracks contiguously, i.e., transfer cost is equal to $\frac{b}{V_s}t_r$, where b is the block size to be transferred, and V_s is the amount of data on one track. For very large blocks, it is likely that several tracks and also tracks in different cylinders are read contiguously. This implies positioning, but we assume that the time used for this is insignificant compared to transfer time. A sector, typical 512 bytes, is the smallest addressable unit on a disk, and this implies that this is the smallest amount of data that can be read. Thus, the total time it takes to transfer one block of size b is:

$$T_B(b) = t_r(1 + \frac{b}{V_s})$$

4.2 Buffer Performance Model

Our buffer model is based on the Bhide, Dan and Dias LRU buffer model (BDD) [2].

A database in the BDD model has a size of N data granules (e.g., pages), partitioned into p partitions. As illustrated in Figure 2, each partition contains a fraction β_i of the data granules, and α_i of the accesses are done to each partition. The distributions *within* each of the partitions are assumed to be uniform, and all accesses are assumed to be independent. We denote a particular partitioning set $\Pi = (\alpha_0, \ldots, \alpha_{p-1}, \beta_0, \ldots, \beta_{p-1})$. For example, for the 80/20 model, $\Pi_{80/20} = (0.8, 0.2, 0.2, 0.8)$.

After n accesses to the database, the number of distinct data granules (pages, objects, or index entries) from partition i that have been accessed is:

$$N^i_{distinct}(n, N, \Pi) = \beta_i N(1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n})$$

The total number of distinct data granules accessed is:

$$N_{distinct}(n, N, \Pi) = \sum_{i=1}^{p} N^i_{distinct}(n, N, \Pi)$$

When the number of accesses n is such that the number of distinct data granules accessed is less than the buffer size B (the number of data granules that fits in the buffer), $\sum_{i=1}^{p} B_i(n) \leq B$, the buffer hit probability for partition i is:

$$P_i(n, \Pi) = 1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n}$$

The overall buffer hit probability is:

$$P(n,\Pi) = \sum_{i=1}^{p} \alpha_i P_i(n,\Pi)$$

The steady state average buffer hit probability can be approximated to the buffer hit ratio when the buffer becomes full, i.e., n is chosen as the largest n that satisfies $\sum_{i=1}^{p} N^i_{distinct}(n,N,\Pi) \leq B$:

$$P_{\text{buf}}(B,N,\Pi) = P(n,\Pi)$$

We have also independently validated these buffer models by the use of simulations, and shown that they have a sufficient accuracy.

4.3 Workload Model

The size of the database in this analysis is S_{DB}, excluding overhead. The number of objects is N_{obj}, which is dependent of the object size.

We assume accesses to objects in the database system to be random, but skewed (some objects are more often accessed than others). We assume it is possible to (logically) partition the objects into partitions, where each partition has a certain size and access probability. Each partition contains a fraction β_i of the data granules, and α_i of the accesses are done to each partition. We will in the comparison use two partitioning sets:

Set	β_0	β_1	β_2	α_0	α_1	α_2
3P1	0.01	0.19	0.80	0.64	0.16	0.20
3P2	0.001	0.049	0.95	0.80	0.19	0.01

In the first partitioning set, we have three partitions. This is an variant of the 80/20 model, but with the 20% hot spot partition further divided, into a 1% hot spot area, a 19% less hot area, and a 80% relatively cold area. The second partitioning set resembles the access pattern close to what we expect it to be in future TODBs, with a large cold set, consisting of old versions.

Unless otherwise noted, results from the analysis are based on calculations using default parameters as summarized in Table 1. Note that even though some of the parameter combinations in the following sections are unlikely to represent the average over time, they can occur in periods, e.g., more write than read operations. In situations like this, when parameter sets differs from the average, which systems traditionally have been tuned against, cost functions can be very helpful to make adaptive self tuning systems.

5 Analytical Modeling of an IPU-TODB

In a temporal database system, it is usually assumed that most accesses will be to the current versions of the objects in the database. To keep these accesses as efficient as possible, and benefit from object clustering, the database is partitioned, with current objects in one partition, and the previous versions in the other partition, in the *historical database*.

When an object is updated in a TODB, the previous version is first moved to the historical database, before the new version is stored in-place in the current database.

Parameter/ Function	Definition	Default Value
V_s	Disk track size	50 KB
t_r	Disk revolution time	$\frac{1}{120}s$
S_{DB}	Database size	8 GB
S_{obj}	Average size of an object	208
P_{write}	Object write probability	0.2
P_{new}	Probability that a write creates a new object	0.2
P_{RC}	Probability that a read is for the current version	0.9
S_P	Page size in page server	4 KB
M_{obuf}	Buffer size	
T_B	Cost (time) of transfer of one block	
$N_{distinct}$	# of distinct object/pages	
P_{buf}	Buffer hit probability	
$T_{readobj}$	Average time to read an object	
$T_{writeobj}$	Average time to write an object	
N_{obj}	Number of objects in the database	$\frac{S_{DB}}{S_{obj}}$
T_S	Cost of writing b bytes sequentially	$\frac{b}{V_s}t_r$
N_{o_page}	Avg. # of objects on each page	$\frac{S_P}{S_{obj}}$
C	Data clustering factor	0.2
T_P	Cost of random read/write a page	
P_{buf_opage}	Page buffer hit probability	
$T_{readpage}$	Average cost of reading a page	
P_{buf_obj}	Object buffer hit probability	
S_{od}	Size of object descriptor	32

Table 1: Summary of system parameters and functions. The horizontal lines separate the general purpose parameters and functions from those specific for IPU-TODB and LO-TODB.

We assume that clustering is not maintained for historical data, so that all objects going historical, i.e., being moved because they are replaced by a new current version, can be written sequentially, something which reduces update costs considerably.

Not all the data in a TODB is temporal, for some of the objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the data can be defined as non-temporal. Old versions of these are not kept, and objects can be updated in-place as in an one-version ODB.

In a traditional page server system, a dedicated disk is usually used for the log. In this way, disk seek is avoided when writing the log, and the log writing can be done efficiently. Having a separate disk is also necessary to handle media failures. We assume the log writing costs are less than the other costs involved, so that when a separate log disk is used, we do not have to consider the cost of log operations in the analysis.

5.1 Clustering

In general, one or more objects are stored on each disk page. To reduce the object retrieval cost, objects are often placed on disk pages in a way that makes it likely that more than one of the objects on a page that is read, will be needed in the near future. This is called clustering.

In our model, we define the clustering factor C as the fraction of an object page that is relevant. If there are N_{o_page} objects on each page, and n of them will be used, the clustering factor (when $N_{o\_page} >= 1$) is:

$$C = \frac{n}{N_{o\_page}}$$

5.2 Object Read Cost

Even with several page requests at a time, and employing an elevator algorithm, the seek time will be significant. The cost of reading or writing a page of size S_P to/from the disk is equal to $T_P = T_B(S_P)$. When reading a page, the page may already be resident in the page buffer. The probability for this is P_{buf_opage}:

$$P_{buf\_opage} = P_{buf}(\frac{M_{obuf}}{S_P + S_{overhead}}, \frac{S_{DB}}{S_P}, \Pi)$$

The average cost of reading a page when taking the buffer into account is:

$$T_{readpage} = (1 - P_{buf\_opage})T_P$$

Assuming a clustering factor of C, the average object retrieval cost from the current partition is:

$$T_{readobj\_cur} = \frac{1}{CN_{o\_page}}T_{readpage}$$

The average object retrieval cost from the historical partition, where we can not assume any clustering is:

$$T_{readobj\_hist} = T_{readpage}$$

We denote the probability that a read operation is for the current version as P_{RC}. The average object retrieval cost is:

$$T_{readobj} = P_{RC}T_{readobj\_cur} + (1 - P_{RC})T_{readobj\_hist}$$

5.3 Object Update Cost

Updating can be done in-place, with write-ahead logging. In that case, a transaction can commit after its log records have been written to disk. Modified pages are not written back immediately, this is done lazily in the background as a part of the buffer replacement and checkpointing. Thus, a page may be modified several times before it is written back, and update costs will be dependent of the checkpoint interval. The checkpoint interval is defined to be the number of objects that can be written between two checkpoints. This number of

written objects, N_{CP}, includes created as well as updated objects. We assume that the buffer is large enough to hold pages written to several times during one checkpoint interval, i.e., we always use a value of N_{CP} where this is true. This is already true in many configurations, and will certainly be valid for future system with large amounts of main memory. N_{CR} of the written objects during a checkpoint period are creations of new objects:

$$N_{CR} = P_{new} N_{CP}$$

Of the objects written during a checkpoint period, $(N_{CP} - N_{CR})$ are updates of existing objects, and the number of *distinct* updated objects is:

$$N_{DU} = N_{distinct}(N_{CP} - N_{CR}, N_{obj})$$

With a sufficiently large checkpoint interval, there will be several writes to a page. The average number of times each object is updated is:

$$N_U = \frac{N_{CP} - N_{CR}}{N_{DU}}$$

During one checkpoint interval, the number of pages in the current partition of the database that is affected is:

$$N_P = \frac{N_{DU}}{N_{o\_page} C}$$

This means that during one checkpoint interval, new versions must be inserted into N_P pages. CN_{o_page} objects on each of these pages have been updated, and each of them have been updated an average of N_U times. To maintain the constraint in the model that all affected pages during one checkpoint interval fits in the buffer, N_P must always be less than the number of pages that fits in the buffer.

For each of the N_P pages, we need to write $N_U C N_{o\_page}$ objects to the historical partition (this includes objects from the page and objects who was not installed into the page before they went historical), install the new current version to the page, and write it back. This will be done in batch, to reduce disk arm movement, and benefit from sequential writing of the historical objects. It is possible to compress historical objects before they are written to the historical partition. The cost of writing b bytes sequentially to disk, assuming the amount of data is large enough to ignore seek time is:

$$T_S(b) = \frac{b}{V_s} t_r$$

When creating a new object, a new page will, on average, be allocated for every N_{o_page} object creation. We assume these pages can be written efficiently, in a mostly sequential way. The total object write related cost during one checkpoint interval is:

$$\begin{aligned} T_T &= \textit{Write to historical partition} \\ &\quad + \textit{Write modified pages} \\ &\quad + \textit{Write new pages} \\ &= T_S(N_P N_U C N_{o\_page} S_{obj}) \\ &\quad + N_P T_P \\ &\quad + T_S(\tfrac{N_{CR}}{N_{o\_page}} S_P) \end{aligned}$$

The average object update cost:

$$T_{writeobj} = \frac{T_T}{N_{CP}}$$

6 Analytical Modeling of an Object Based LO-TODB

The LO-TODB modeled in this section operates according to the description in Section 3.

6.1 Object Read Cost

Similarly to the IPU-TODB, we need two disks to be able to handle media failures. In the log-only system, we can use the disks in a mirroring configuration, which doubles the read bandwidth. The average cost of reading an object from disk when we have parallel, independent read operations from the two disks:

$$T_{readobj\_disk} = (T_B(S_{obj}))/2$$

When reading an object, the object may already be resident in the object buffer. The probability of this is P_{buf_obj}:

$$P_{buf\_obj} = P_{buf}(\frac{M_{buf}}{S_{obj} + S_{overhead}}, N_{obj})$$

Taking into account the object buffer, the average object read cost is:

$$T_{readobj} = (1 - P_{buf\_obj})T_{readobj\_disk}$$

The assumption that only one object is read from the disk between each disk seek is actually a very conservative estimate. It does not take into account prefetching, and more important, it does not take into account disk read ahead. In practice, dynamic reclustering will also improve read performance considerably. Without doing a thorough analysis of this aspect, we can get an idea of how much we can gain from this by using the previous cost functions, and assuming that from each random read operation to retrieve an object, we can get n additional objects "cheap". We assume that the disk employs read ahead, so that when we do a read operation, some of the following data will also be read. If doing a disk read shortly after, this data will still be in the cache on the disk, and can be retrieved without an additional disk seek operation. In this way, the TODB does not even have to do any implicit prefetching. If we assume n to be small, so that the total data $D = nS_{obj}$ is small enough to expect it to be cached by read ahead, we can approximate the read cost in this case to be:

$$T_{readobj\_reclustered}(n) = T_{readobj}/n$$

In this paper we will in general assume the worst case, with $n = 1$, but we will in Section 7.5 see how a more realistic value of n will affect performance.

6.2 Object Update Cost

When writing to the log, this is done sequentially, with large blocks (segments). The object write operation in a LO-TODB is to put the object into the log, together with its OD (Section 3):

$$T_{writeobj} = T_S(S_{od} + S_{obj})$$

Writing the OD together with the object is done to avoid synchronous updates of the OIDX, because in a LO-TODB we do not have a separate log to write this information.

7 A Comparison of Performance

We have now derived the cost functions necessary to calculate the average object storage and retrieval costs with different system parameters and access patterns. We will in this section study how different values of these parameters affects the access costs. Optimal parameter values are dependent of the mix of updates and lookup, and they should be studied together. If we denote the probability that an operation is a write as P_{write}, the average access cost is the average of the cost of all object read and write operations:

$$T_{\text{access}} = (1 - P_{\text{write}})T_{\text{readobj}} + P_{\text{write}}T_{\text{writeobj}}$$

We will also use speedup as a metric. If we denote the average object access time for an IPU-TODB as $T^{\text{IPU}}_{\text{access}}$, and the average object access time for an LO-TODB as $T^{\text{LO}}_{\text{access}}$, we can calculate speedup as:

$$Speedup = \frac{T^{\text{IPU}}_{\text{access}}}{T^{\text{LO}}_{\text{access}}}$$

A speedup less than 1.0 means that with the given parameters, an IPU-TODB will perform best. A speedup greater than 1.0, means that an LO-TODB will perform best. On the figures, we have plotted the 1.0 line to make it easy to see under which conditions each of the two approaches have the best performance.

The memory size on the figures in this paper, is the memory size relative to database size.

7.1 Object Access Cost

Figure 3 shows the average object access costs with different access patterns. We see here that the access pattern affects very much under what conditions the different approaches perform best.

7.2 The Effect of Different Object Sizes

The average object size is an important parameter. In this study, we used a default object size of 208 bytes, which is close to size which we have observed used in many other ODB performance studies, for example at the University of Wisconsin, where they have typically use an object size of 200 bytes.[3]

[3]Our chosen value, 208 bytes, is the tuple size in the well known Wisconsin Benchmark for relational database systems.

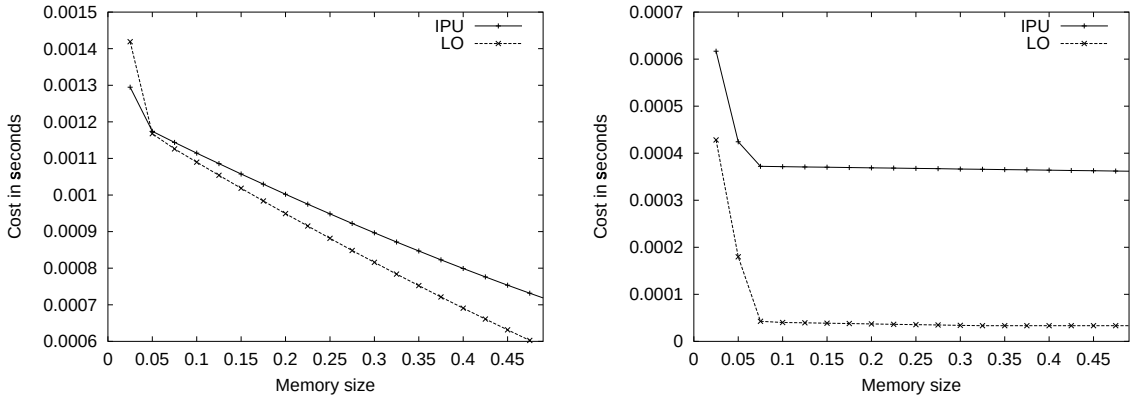

Figure 3: Object access cost. Access pattern 3P1 to the left and 3P2 to the right.

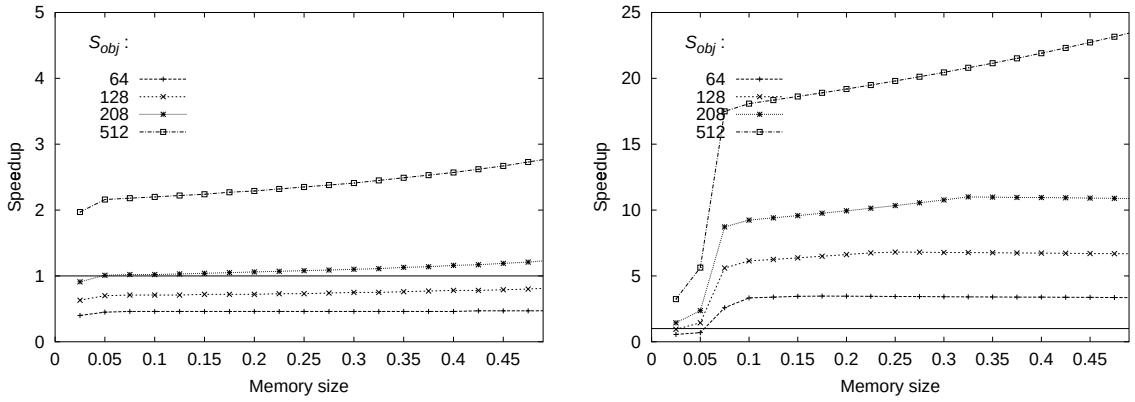

Figure 4: Speedup with different object sizes. Access pattern 3P1 to the left and 3P2 to the right.

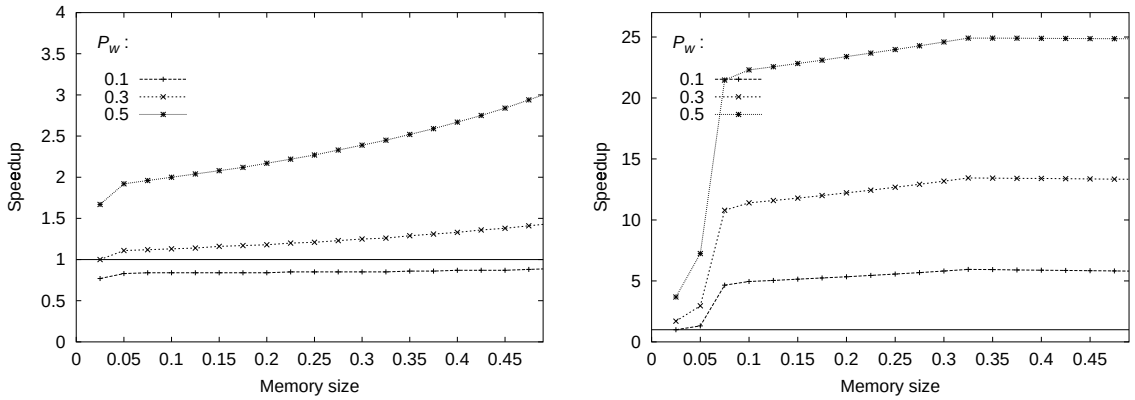

Figure 5: Speedup with different update ratios. Access pattern 3P1 to the left and 3P2 to the right.

Figure 4 illustrates the speedup with different object sizes. It shows very well what can be expected, and it is important to note that even the largest object size used here, 512 bytes, is not really a very large object! The average object size is increasing as a result of new application areas and cheaper storage, which means that we can expect a better speedup from using an LO-TODB in the future.

7.3 The Effect of Different Update Ratios

In this analysis, we have used $P_{write} = 0.2$ as the default value for the fraction of operations being write operations. In periods, and in some application areas, we will have a higher value of $P_{write} = 0.2$. Figure 5 shows how different update ratios affects the speedup. We see that with a higher value of P_{write}, the speedup is considerably higher than for small values of P_{write}.

7.4 The Effect of Different Clustering Factors

In most page server ODBs, it is possible to advice the system on how to cluster objects on the pages. Unfortunately, usually only the initial clustering can be specified, reclustering is often impossible. In a multiuser system, with complex and dynamic workloads, it is commonly difficult to find a good clustering. The default clustering factor in our analysis is 0.25, which favors the IPU-TODB approach.

Figure 6 shows how the clustering factor in IPU-TODBs affect their performance, and their relative performance to LO-TODB. This shows well how much IPU-TODBs depend on good clustering. This is an important point. In practice, with different applications accessing a database, it is difficult to get a good clustering factor. In a study by Tsangaris and Naughton [14], all practical clustering algorithms results in an average clustering which is less than this value, for the clustering algorithms and workloads in this study, C had values between 0.25 and 0.1.[4] In a real world application, clustering will often be worse. That study

[4]Tsangaris and Naughton use the metric *expansion factor (EF)*, where $EF = 1/C$.

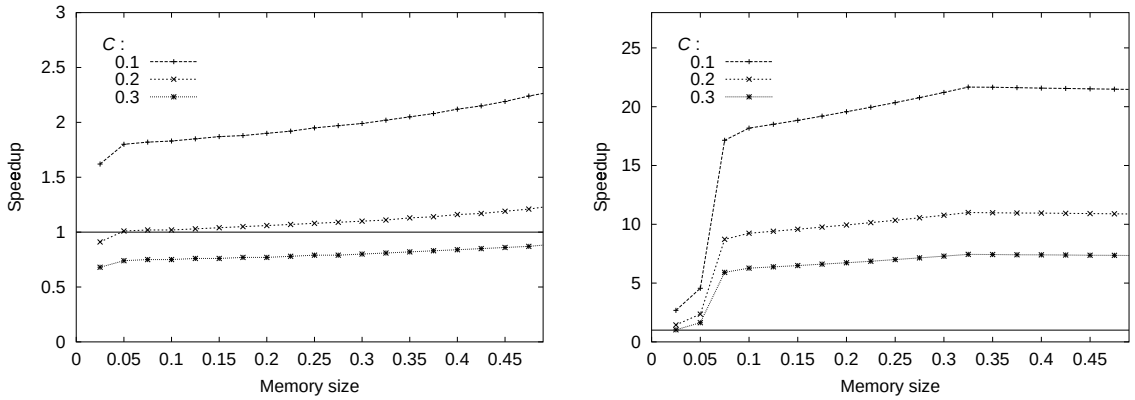

Figure 6: Speedup with different clustering factors. Access pattern 3P1 to the left and 3P2 to the right.

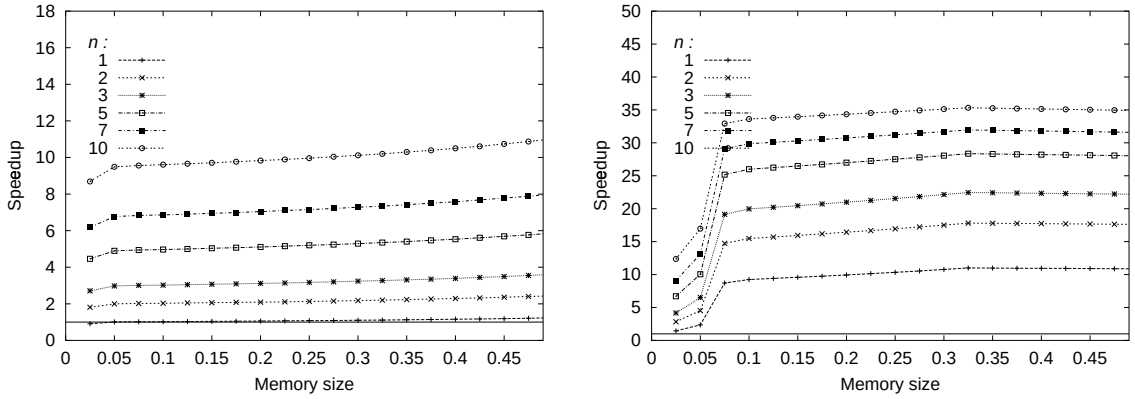

Figure 7: Speedup, reclustered. Access pattern 3P1 to the left and 3P2 to the right.

was done with a 4 KB page size. With a larger page size, C would probably decrease because of a higher degree of false sharing.

As is evident for the figure, if we consider a more likely clustering factor, e.g., less than 0.20, an LO-ODB will perform better than an IPU-TODB for both access patterns, even with a much smaller amount of main memory available.

7.5 The Effect of Reclustering and Disk Read Ahead in an LO-TODB

Until this point, we have assumed no clustering in the LO-TODB, with only one read object on each disk access. It is very likely that it is possible to achieve better than this. This can be achieved by writing related objects at the same time, and by "intelligent" cleaning and reorganization of the segments.

By using a relatively high default value for the clustering factor in an IPU-TODB, we have already assumed it is well clustered. By using the modified read cost in the LO read object cost function with different values of n, we get an indication of how much can be gained. Figure 7 shows speedup with different values of n with the different access patterns.

7.6 The Impact of Cleaning and Reorganization

The major disadvantage of the log-only approach has been reported to be segment cleaning cost. Normally, the cleaner is running in the background, and only employing idle resources. However, under certain conditions, its cost will be significant, for example in a system with no idle periods, or when the disk utilization is too high. As a result, the cleaning cost can be devastating in a non-temporal LO-ODB. In a LO-TODB, on the other hand, we expect the cleaning cost to be much lower, because we keep old versions of most objects. Additionally, the gain from reclustering during cleaning might well outweight the disadvantages of the cleaning.

8 Conclusions

The log-only approach is particularly interesting for transaction time ODBs. We have shown by the use of analytical cost models that with the workload we expect to be typical for TODBs, the log-only approach is highly competitive with the traditional in-place update approach. We expect the benefits of the log-only approach to be even more interesting in the future, with increasing amounts of main memory available and increasing object sizes. We should also again emphasize that the cost model for the log-only TODB that has been developed in this paper is very conservative. We expect both clustering and buffer hit ratio to be better in practice, resulting in a better performance.

In addition to good performance on small objects, which have been shown by the analysis in this paper, there are other features in a log-only system that makes it even more interesting. This includes fast recovery and flexible large object chunk size. The importance of this should not be underestimated.

The work presented in this paper has been done in the context of the Vagabond TODB, but it should be equally applicable in the context of an object-relational database system.

References

[1] M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2), 1976.

[2] A. K. Bhide, A. Dan, and D. M. Dias. A simple analysis of the LRU buffer policy and its relationship to buffer warm-up transient. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[3] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2), 1981.

[4] D. Hulse and A. Dearle. A log-structured persistent store. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[5] P. Muth, P. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, 1998.

[6] K. Nørvåg. The Persistent Cache: Improving OID indexing in temporal object-oriented database systems. In *Proceedings of the 25th VLDB Conference*, 1999.

[7] K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.

[8] K. Nørvåg and K. Bratbergsengen. Write optimized object-oriented database systems. In *Proceedings of the XVII International Conference of the Chilean Computer Science Society, SCCC'97*, 1997.

[9] K. Nørvåg and K. Bratbergsengen. Optimizing OID indexing cost in temporal object-oriented database systems. In *Proceedings of the 5th International Conference on Foundations of Data Organization, FODO'98*, 1998.

[10] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 1996.

[11] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, 1991.

[12] V. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, 1992.

[13] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th VLDB Conference*, 1987.

[14] M. Tsangaris and J. Naughton. On the performance of object clustering techniques. In *Proceedings of SIGMOD'92*, 1992.