Optimizing OID Indexing Cost in Temporal Object-Oriented Database Systems

Kjetil Nørvåg and Kjell Bratbergsengen
Department of Computer and Information Science
Norwegian University of Science and Technology
7034 Trondheim, Norway
{noervaag,kjellb}@idi.ntnu.no

Abstract

In object-oriented database systems (OODB) with logical OIDs, an OID index (OIDX) is needed to map from OID to the physical location of the object. In a transaction time temporal OODB, the OIDX should also index the object versions. In this case, the index entries, which we call *object descriptors* (OD), also include the commit timestamp of the transaction that created the object version. In this report, we develop an analytical model for OIDX access costs in temporal OODBs. The model includes the index page buffer as well as an OD cache. We use this model to study access cost and optimal use of memory for index page buffer and OD cache, with different access patterns. The results show that 1) the OIDX access cost can be high, and can easy become a bottleneck in large temporal OODBs, 2) the optimal OD cache size can be relatively large, and 3) the gain from using an optimal size is considerable, and because access pattern in a database system can be very dynamic, the system should be able to detect this, and tune its memory use according to this. The cost models in this report can be of valuable use for optimizers and automatic tuning tools in temporal OODBs. The primary context of this report is OID indexing in a temporal OODB, but the results are also relevant in the context of general secondary index access cost and index entry caching.

Keywords: Index organization, temporal database systems, object-oriented database systems, buffer management

1 Introduction

In a traditional object-oriented database system (OODB), updating an object makes the old version unaccessible. In a temporal object-oriented database system (TOODB), on the other hand, updating an object simply creates a new version of the object, the old version is still accessible. In a transaction time TOODB, which is the context of this report, a system maintained timestamp is associated with every object version. This timestamp is the commit time of the transaction that created this version of the object.[1]

In an OODB, an object is uniquely identified by an object identifier (OID). The OID can be physical, which means that the disk page of the object is given directly from the OID, or logical, which means that an OID index (OIDX) is needed to map from logical OID to the

[1] In the other common category of temporal database systems, valid time database system, a time interval is associated with every object, denoting the time interval which the object is valid in the modeled world.

physical location of the object. A physical OID scheme is inflexible, and in a temporal OODB, logical OIDs is the only reasonable alternative, because of objects being moved. The entries in the OIDX, the *object descriptors* (OD)s, contain administrative information, including information to do the mapping from logical OID to physical address. An object is accessed via its OID, hence, lookup in the OIDX have to be as efficient as possible. The OIDX can be quite large, typical in the order of 20% of the size of the database itself [4]. This means that in general, only a small part of the OID index fits in main memory, and that OID index retrieval can become a bottleneck if efficient access and buffering strategies are not applied. An important difference between OIDX management in non-temporal and versioned OODBs, is that with no versioning, the OIDX needs only to be updated when an object is created, which can be done efficiently as an efficient append-only operation. In a TOODB however, the OIDX must be updated every time an object is updated. An object update creates a new object version, without deleting the previous version, hence, a new OD for the new version have to be inserted into the OIDX. The index pages will in general have low locality (the unique part of an OID is usually an integer that will always be assigned monotonic increasing values), and index updates might become a serious bottleneck in a TOODB.

We will in this report study the cost of OIDX accesses, inserts as well as lookups, and the use of buffering to speed up the OID mapping. Our approach is based on analytical modeling. Compared to simulations, another common used approach, analytical modeling has the advantage that it is easy to change parameters, to be able to explore larger areas of the parameter space. With an analytical model, cost and optimal values for configurable parameters can be obtained fast. This is important for automatic system tuning and query planning. As systems get larger, more complex, and more flexible, automatic tuning is needed, or at least support to a greater extent than today. One way to accomplish this, is to use analytical system models to compute optimal configuration parameters, e.g., buffer sizes, so that the systems can adapt to changing workloads, and always give as high performance as possible. Cost models are also useful for query planning, to choose the best query plans, and the results from this report can also be used for system design. It should also be noted that even though our primary context for this report is OID indexing, the results are also relevant to entry access cost and entry caching for general secondary indexes.

The organization of the rest of the report is as follows. In Section 2 we give an overview of related work. In Section 3 we describe object and index management in TOODBs. In Section 4 we describe our index entry access model. In Section 5 we present our index page access model, the Bhide, Dan and Dias LRU buffer model [2], which our hierarchical index buffer model is based on. In Section 6 we develop an the OID access cost model, and in Section 7 we use this cost model to study how different memory sizes, index sizes, access patterns and disk page sizes affect the performance. Finally, in Section 8, we conclude the report and outline issues for further research.

2 Related Work

Several cost models for OODBs have been developed, for example [1, 6], but they do not include buffer and OID mapping cost. In a previous report and paper, we have studied index lookup cost and optimal use of memory in a one-version OODB [11, 12]. The buffer and tree models have been compared with simulation results. Detailed results from the simulations with different index sizes, buffer sizes, index page fanout, and access patterns can be found in [12].

Temporal database systems are in general still an immature technology, and in the case of transaction time TOODBs, we are only aware of one prototype[2] that have temporal OID indexing [13].

3 TOODB Object and Index Management

We start with a description of how OID indexing and version management can be done in a TOODB. This brief outline is not based on any existing system, but the design is close enough to current OODBs to make it possible to integrate if desired.

3.1 Temporal OID Indexing

In a traditional OODB, the OIDX is usually realized as a hash file or a B-tree, with ODs as entries, and using the OID as the key. In a TOODB, we have more than one version of some of the objects, and we need to be able to access current as well as old versions efficiently. If access is mostly reading current objects, it is efficient to have two indexes, one with ODs representing the current version of the objects, and one with ODs representing historical objects (i.e, previous versions). The problem with this approach, is that every time a new version is created, we have to update *two* indexes. A second approach, is to use a linked list of versions for each object. If accesses are mostly of the type "get all versions of an object with OID i", an efficient alternative is to use a linked list of versions for each object. . However, access to a particular version, valid at time t, is very costly with this approach, because we have to traverse the object chain.

Our approach to indexing is to have *one* index structure, containing all ODs, current as well as previous versions. While several efficient multiversion access methods exist, e.g., TSB-tree [7] and LHAM [10], they are not suitable for our purpose. We will never have search for a (consecutive) range of OIDs, OID search will always be for *perfect match*, and most of them are assumed to be to the current version. TSB-trees provides more flexibility than needed, e.g., combined key range and time range search, which implies an extra cost, while LHAM can have a high lookup cost when the current version is to be searched for.

In this report, we assume one OD for each object version, stored in a B$^+$-tree. We include the commit time in the OD, and use the concatenation of OID and time, $OID||TIME$, as the index key. In a transaction time TOODB, $TIME$ is the commit time for each version that is stored. To increase the number of ODs in an index node, prefix compression on OIDs can be employed. When a new object is created, i.e., a new OID allocated, its OD is appended to the index tree as is done in the case of the Monotonic B$^+$-tree [5]. In the Monotonic B$^+$-tree, all new entries is appended to the tree, and we do not distribute the entries over the old node and the new, as is done in B$^+$-trees. When an object is updated, the OD for the new version have to be inserted into the tree.

It should be noted that this OIDX is inefficient for many typical temporal queries. As a result, additional secondary indexes can be needed, of which both TSB-tree and LHAM are good candidates.

[2]Support for versioning exists in most OODBs, but not temporal management, indexing, and operations.

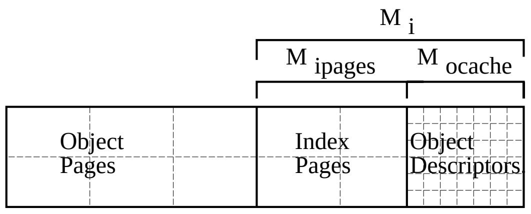


Figure 1: Object page buffer, index page buffer, and OD cache in a page server TOODB.

3.2 Temporal Object Management

In a one-version OODB, space is allocated for an object when it is created, and updates to the objects are done in-place. This implies that after an object update, the previous version of the object is not available. The physical location of the new version is the same as the previous version, hence, the OIDX needs only to be updated when objects are created and when they are deleted.

In a TOODB, 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,[3] 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 TOODB, the previous version is first moved to the historical database, before the new version is stored in-place in the current database. The OIDX needs to be updated *every time an object is updated*. As long as the modified ODs are written to the log before commit, we do not need to update the OIDX itself immediately. This is done in the background, and can be postponed until the second checkpoint after the OD have been written to the log. Index pages will be written to disk either because of checkpointing, or because of buffer replacement.

Versioning will not be needed for all the data in a TOODB. To improve efficiency, the system can be made aware of this. In this way, some of the data can be defined as non-versioned. Old versions of these are not kept, and objects can be updated in-place as in a one-version OODB, and the costly OIDX update is not needed when the object is modified. This is an important point, using an OODB which efficiently supports temporal storage, should not reduce the performance of applications that do not utilize this feature.

3.3 Index Page Buffer and OD Cache

To reduce disk I/O, the most recently used *index pages* are kept in an *index page buffer*. OIDX pages will in general have low locality, and to increase the probability of finding a certain OD needed for a mapping from OID to physical address, it is also possible to keep the most recently used *index entries* (the ODs) in a separate OD cache, as is done in the Shore OODB [9]. With low locality on index pages, a separate OD cache utilizes memory better, space is not wasted on large pages where only small parts of them will be used. The size of the OD cache can be fixed (set at system startup time), or adaptive. The buffer in a page server OODB is illustrated on Figure 1. In the rest of this report, we will denote the index

[3] It is also possible that in a TOODB application, a good object clustering includes historical objects as well as current objects. This should be studied further, but does not have any implications to the results studied here, all updates to objects will necessarily necessitate allocations of space for the new object, and an OIDX update.

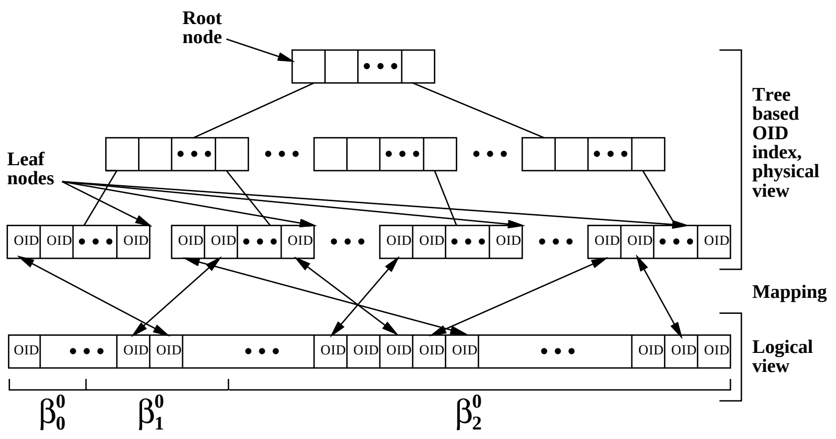


Figure 2: OID index. The lower part shows the index from a logical view, the upper part is as an index tree, which is how it is realized physically. We have indicated with arrays how the entries are distributed over the leaf nodes.

page buffer size as M_{ipages}, the OD cache size as M_{ocache}, and the sum of these as M_i, as illustrated on the figure.

4 Index Entry Access Model

We assume accesses to objects in the database system to be random, but skewed (some objects are more often accessed than others). We further assume it is possible to (logically) partition the range of OIDs into partitions, where each partitions has a certain size and access probability. This is illustrated on the bottom of Figure 2 (Note that this is not how it is stored on disk, this is just a model of accesses). We consider a database in a stable condition, with a total of N_{objver} objects versions (and hence, N_{objver} index entries). Note that with the OIDX described in Section 3.1, performance is not dependent of the number of existing versions of an object, only the total number of versions in the database.

In many analysis and simulations, the 80/20 model is applied, where 80% of the accesses go to 20% of the database. While this is satisfactory for analysis of some problems, it has a major shortcoming when used to estimate the number of distinct objects to be accessed. When applied, it gives a much higher number of distinct accessed objects than in a real system. The reason is that for most applications, inside the hot spot area (20% in this case), there is an even hotter and smaller area, with a much higher access probability. This is even more important for a temporal database. Most of the accesses will be to a small number of the current versions. With a large number of previous versions, this hot spot area will be much smaller and "hotter" than the one in a typical "traditional" database. This has to be reflected in the model.

5 Buffer Performance Model

The buffer model presented in this section has been described earlier [11, 12], but is included here to make this report self contained.

5.1 Index Page Access Model

As noted, we can assume low locality in index pages. Because of the way OIDs are generated, entries from a certain partition are not clustered in the index. This is illustrated in Figure 2, where a leaf node containing index entries contains unrelated entries from different partitions. This means that the access pattern for the leaf nodes is different from the access pattern to the database from a logical view.

We use the initial index entry partitioning (the index entry access pattern) as basis for deriving the index page partitioning (the index page access pattern). With a totally unclustered index, the index entries from the different partitions are distributed over the leaf nodes with binomial distribution. To simplify this analysis, we make some approximations and assumptions: The second most hot area has a number of entries sufficiently larger than the number of leaf nodes, i.e., $\beta_1^0 > \frac{1}{F}$. This is a reasonable assumption, with a page size of 8 KB, this gives $\beta_1^0 > 0.002$. We assume that the accesses to entries not belonging to the hottest hot spot can be modeled as random and uniform, and it is the hottest hot spot area alone that decides the index page access pattern. Further, we approximate the binomial distribution by assuming that the index entries are distributed over the leaf nodes with uniform distribution. Simulation results show that the error is acceptable.

We consider two cases: 1) the number of hot spot entries is smaller than the number of leaf nodes, $\beta_0^0 N_{\text{objver}} < N_{\text{tree}}^0$, and 2) the number of hot spot entries is larger than the number of leaf nodes, $\beta_0^0 N_{\text{objver}} > N_{\text{tree}}^0$.

Case 1: The number of hot spot entries is smaller than the number of leaf nodes. When the number of objects in the hot spot area is less than the number of leaf nodes, pages belong to one of two partitions: Those that have a hot spot entry, and those which have not:

$$\begin{aligned} \beta_{L0} &= \beta_0^0 N_{\text{objver}}/N_{\text{tree}}^0 & \alpha_{L0} &= \alpha_0^0 + \beta_{L0} \textstyle\sum_{i=1}^{p} \alpha_i^0 \\ \beta_{L1} &= 1 - \beta_{L0} & \alpha_{L1} &= 1 - \alpha_{L0} \end{aligned}$$

Case 2: The number of hot spot entries is larger than the number of leaf nodes. In the case where there is one or more hot entry on each page, we will with a uniform distribution over the pages have some of the pages with one hot index entry more than the others. Especially in the case where there are few hot index entries on each pages, it is important to capture this fact in the model. We now have two partitions, one with the pages containing that extra index entry, and the other with those pages that do not:

$$\begin{aligned} \beta_{L0} &= (\beta_0^0 N_{\text{objver}} - \lfloor \beta_0^0 N_{\text{objver}}/N_{\text{tree}}^0 \rfloor N_{\text{tree}}^0)/N_{\text{tree}}^0 \\ \beta_{L1} &= 1 - \beta_{L0} \\ \alpha_{L0} &= \beta_{L0} N_{\text{tree}}^0 \frac{\lceil \beta_0^0 N_{\text{objver}}/N_{\text{tree}}^0 \rceil}{\beta_0^0 N_{\text{objver}}} \\ \alpha_{L1} &= 1 - \alpha_{L0} \end{aligned}$$

With an increasing number of hot spot index entries on each page, the access will get more and more uniform.

We will in the following use Π_A as short for the data entry access partitioning, and Π_L as short for the leaf node access partitioning, where Π_L is calculated from Π_A as described above.

5.2 The BDD LRU Buffer Model

In our analysis, we need to estimate the buffer hit probability in an LRU managed buffer. We do this with the Bhide, Dan and Dias LRU buffer model (BDD) LRU buffer model [2]. We will only briefly explain the model in this section, the derivation and details behind the equations can be found in [2]. A database in the BDD model has of size N data granules (pages or objects), partitioned into p partitions. Each partition contains 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. All accesses are assumed to be independent. We denote this particular partitioning as Π.

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:

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

and 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, $\sum_{i=1}^{p} B_i(n) \leq B$, the buffer hit probability for partition i is:

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

and 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 $\sum_{i=1}^{p} N^i_{distinct}(n, N, \Pi) \leq B$, where B in this case is the number of data granules that fits in the buffer:

$$P_{\text{buf}}(B, N, \Pi) = P(n, \Pi) \tag{1}$$

5.3 General Index Buffer Model

In the previous section, we presented the BDD LRU buffer model for independent, non-hierarchical, access. Modeling buffer for hierarchical access is more complicated. Even though searches to the leaf page can be considered to be random and independent, nodes accessed during traversal of the tree are *not* independent. We will in this section present a general index buffer model, where the granularity for access is a page.

In the generic tree used in this model, the size of one index page is S_P, and the size of one index entry S_{ie}. If we define U as the utilization (typically 0.69 for a B-tree), this give a fanout $F = \lfloor U S_P / S_{\text{ie}} \rfloor$ and with a database consisting of N_{objver} objects to be indexed, the number of leaf nodes is $N^0_{\text{tree}} \approx \frac{N_{\text{objver}}}{U \lfloor S_P / S_{\text{ie}} \rfloor}$.

Our approach to approximate the buffer hit probability, is based on the obvious observation that *each level* in the tree is accessed with the *same probability* (assuming traversal from root to leaf on every search). Thus, with a tree where the index nodes has a fanout F, the number of levels in the tree is $H = 1 + \lceil \log_F N^0_{\text{tree}} \rceil$. We initially treat each level in the tree as one partition, thus, initially we have H partitions. Each of these partitions are of size N^i_{tree}, where N^i_{tree} is the number of index pages on level i in the tree. The access probability is $\frac{1}{H}$ for each partition.

To account for hot spots, we further divide the leaf page partition into p' partitions, each with a fraction of β_{Li} of the leaf nodes, and access probability α_{Li} relative to the other leaf page partitions. Thus, in a "global" view, each of these partitions have size $\beta_{Li} N^0_{\text{tree}}$ and access probability $\frac{\alpha_{Li}}{H}$. In total, we have $p = p' + (H - 1)$ partitions. The hot spots at the leaf page level make access to nodes on upper levels non-uniform, but as long as the fanout is sufficiently large, and the hot spot areas are not too narrow, we can treat accesses to nodes on upper levels as uniformly distributed within each level. With the modifications described, a tree of height H, and p' leaf page partitions, the equations for α_i and β_i (access probability and fractional size of each partition) becomes:

$$\alpha_i = \begin{cases} \frac{\alpha_{Li}}{H} \\ \frac{1}{H} \end{cases} \quad \textit{and } \beta_i = \begin{cases} \frac{\beta_{Li} N^0_{\text{tree}}}{N_{\text{tree}}} & \text{if } i < p' \\ \frac{N^i_{\text{tree}}}{N_{\text{tree}}} & \text{if } p' \le i < p \end{cases}$$

We denote the number of leaf nodes as N^0_{tree}. The total number of pages in the tree, N_{tree}, can then be calculated as:

$$N_{\text{tree}} = \sum_{i=0}^{H-1} N^i_{\text{tree}} \quad \text{where} \quad N^i_{tree} = \left\lceil \frac{N^{i-1}_{\text{tree}}}{F} \right\rceil \tag{2}$$

We denote the overall buffer probability, by using the BDD buffer hit probability equation (Equation (1)) with α_i and β_i as defined above as $P_{\text{buf\_ipage}}(B, N_{\text{tree}})$ where B is the buffer size, and N_{tree} is the total number of index pages in the tree.

6 Index Access Cost

Analytical modeling in database research has mostly focused on I/O costs. This is the most significant cost factor, and in reasonable implementations, the CPU processing should go in parallel with I/O transfer making the CPU cost "invisible". With increasing amounts of main memory available, this is not necessarily correct, but CPU costs can easily be incorporated into analytic models, and hence we consider it as an orthogonal issue to the one discussed in this report (though it should be noted, that CPU cost should not affect the qualitative results in this report). A more important aspect of the increasing amount of main memory, however, is that buffer characteristics become more important, hence, the increased buffer space available must be reflected in the models.

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 is directly proportional to the block size, and is equivalent to reading disk tracks contiguously, e.g., 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. Thus, the total time it takes to transfer one block is $T_b(b) = T_{start} + T_{transfer} = t_r + \frac{b}{V_s} t_r$.

The time to read or write a random index page is $T_P = T_b(S_P)$, where S_P is the index page size. In this report, we do not consider the cost of reading and writing the objects themselves, or log operations. Those costs are independent of the indexing costs, usually done on separate disks, and are issues orthogonal to the ones studied in this report.

6.1 Buffer Hit Probabilities

A certain amount of memory, M_{ipages}, is reserved for the index page buffer, and M_{ocache}, is reserved for the OD cache. If we assume the size of each OD is S_{od}, and an overhead of S_{oh} bytes is needed for each entry in the OD cache, the number of entries that fits in the OD cache is approximately $N_{\text{ocache}} \approx \frac{M_{\text{ocache}}}{(S_{\text{od}} + S_{\text{oh}})}$. Accesses to the OD cache can be assumed to follow the assumptions behind the BDD model, they are independent random requests, and by applying this model with object entries as data granules, we estimate that the probability of an OD cache hit is $P_{\text{ocache}} = P_{\text{buf}}(N_{\text{ocache}}, N_{\text{objver}}, \Pi_A)$.

Accesses to the index page buffer, on the other hand, follows the general index buffer model (Section 5.3). The number of index pages that fits in the buffer is approximately $N_{\text{ibuf}} \approx \frac{M_{\text{ipages}}}{(S_{\text{P}} + S_{\text{oh}})}$, and the probability of an index page buffer hit is $P_{\text{buf\_ipage}}(N_{\text{ibuf}}, N_{\text{tree}})$.

The results in the following analysis are highly dependent of the amount of overhead S_{oh} needed for each entry. Of course, with a minimal overhead, it would always be beneficial to use as much as possible of the total index memory as an OE cache. The most reasonable way to implement it, and at the same time keep the CPU cost low, is to use an hash table to provide fast access to the entries. In that case, approximately two pointers are needed for each entry on average. We also need some additional data structures to do the buffer management. Even though we base the analysis on an LRU buffer, a clock algorithm will probably be used. It has performance close to LRU [3], but has less storage overhead, only one bit is needed for each entry. This is small enough to ignore in this analysis.

6.2 Index Lookup Cost

With a probability of P_{ocache}, the OID entry requested is already in the OD cache, but for $(1 - P_{\text{ocache}})$ of the requests, we have to access the index pages, and one or more disk accesses might be needed. The probability of a given index page being in memory is *on average* $P_{\text{buf\_ipage}}$. To access an entry (OD) in an index page, it is necessary to traverse the H levels from the root to a leaf page. To do this, we need $(1 - P_{\text{buf\_ipage}})H$ disk accesses. The average cost of an index lookup is $T_{\text{lookup}} = (1 - P_{\text{ocache}})(1 - P_{\text{buf\_ipage}})HT_P$.

6.3 Index Update Cost

We do not need to update the index pages in the OIDX immediately. This is done in the background, and can be postponed, increasing the probability that several updates can be done to the index page before it is written back. We calculate the average index update cost as the total index update cost during one checkpoint interval, divided on the number of index updates. In this context, we define the checkpoint interval to be the number of objects that can be written between two checkpoints. The number of written objects, N_{CP}, includes created as well as updated objects. $P_{new}N_{CP}$ of the written objects are creations of new objects, and $(1 - P_{new})N_{CP}$ of the written objects are updates of existing objects. We assume that memory is large enough to keep all dirty ODs through one checkpoint interval, and that delete and compacting pages can be done in background.

Creation of new object descriptors. New object descriptors are created when new objects are created. The number of created objects is $N_{CR} = P_{new}N_{CP}$. When new objects are created, their ODs are appended to the index, and we have clustered updates. This contributes to $N_n^0 = \frac{N_{CR}}{\lfloor S_P/S_{\mathrm{od}} \rfloor}$ created leaf pages. This is a subtree in the index tree, of height H_s, with $S_n = \sum_{i=0}^{H_s-1} N_n^i$ pages (Equation 2). The total cost of creating these object descriptors is the cost of writing S_n index pages to the disk, no installation read is needed for these pages. Assuming that the disk is not too fragmented, these pages can be written in one operation, most of them sequentially:

$$T_{writenew} = T_b(S_n S_P)$$

Modification of existing object descriptors. When an object is updated, a new object version is created, and a new OD has to be inserted into the OIDX. The number of updated objects is $N_U = N_{CP} - N_{CR}$. Updating the index involves a page installation read, where the page where the last (current) version resides is read from disk, if the page is not already in the buffer. The cost of this is T_{lookup} for each *distinct* object modified. The number of distinct updated objects is:

$$N_{DU} = N_{distinct}(N_{CP} - N_{CR}, N_{\mathrm{objver}}, \Pi_A)$$

However, as noted in Section 3.2, not all objects in a TOODB are versioned. We denote the fraction of the data accesses going to versioned objects as $P_{versioned}$. Only updates of these objects alter the OIDX, updates of non-versioned objects will done in-place, thus the number of distinct updated versioned objects is:

$$N_{DU}^V = P_{versioned} N_{DU}$$

The number of leaf pages to be accessed as a part of the installation read:

$$N_m = N_{distinct}(N_{DU}^V, N_{tree}^0, \Pi_L)$$

If there is space for the new OD in the leaf node, it can be inserted there, and the node can be written back. If there is no space in the node, the node is split, a process done recursively, possibly to the root. If a node is split, the parent node has to be updated as well (except in the case when the root node is split, in this case, the height of the tree is increased with one level). Because of the possibility of page splits, determining the update cost is difficult. With sufficiently many entries on each index node, the probability of page split is small enough to be neglected [14]. However, for some pages, there are more than one insertion to that page (possibly generated by several updates to one object during one checkpoint interval, remember that *each update creates a new entry to be inserted into the OIDX*). Thus, we include the page split in our cost functions. According to Loomis [8], the probability of a split in a B-tree of order m is less than $\frac{1}{\lceil m/2 \rceil - 1}$, so we approximate $P_{split} \approx \frac{1}{\lceil (S_P/S_{\mathrm{od}})/2 \rceil - 1}$. For each split, the new page needs to be written back, as well as the updated parent node. However, note that there may be several splits affecting one parent node, in this case, it needs only be written back once. The resulting total write back cost is:

$$T_{writeback} = (N_m + N_m 2P_{split})T_P = N_m(1 + 2P_{split})T_P$$

If the checkpoint interval is sufficiently large, it is more efficient to read the complete index, update the index nodes, and write it back (if memory is not large enough, this is done in segments). This will be very efficient, as the reading and writing will be sequential. The cost of this is:

$$T_{readwrite\_all} = 2T_b(N_{tree}S_P)$$

The total cost of index update during one checkpoint interval:

$$T_{update\_total} = T_{writenew} + \min(N_m T_{lookup} + T_{writeback}, T_{readwrite\_all})$$

The average index update cost per object:

$$T_{update} = \frac{T_{update\_total}}{N_{CP}}$$

7 Analytical Study

We have now derived the cost functions necessary to calculate the average cost of OIDX access under different system parameters and access patterns. We will in the following sections study in detail how different values for these parameters affects the access cost, and how the optimal OD cache size changes as well. The mix of updates and lookups to the OIDX affects the optimal parameter values, and they should be studied together. If we denote the probability that an object operation is a write, as P_{write}, the average index access cost is the average of the cost of all index lookup and index update operations:

$$T_{acccess} = (1 - P_{write})T_{lookup} + P_{write}T_{update}$$

We define optimal OD cache size as the size of the OD cache that, given a certain amount of memory available for index pages and OD cache,[4] gives the lowest average index access cost, i.e., find the M_{ocache} that minimizes $T_{acccess}$, given the invariant:

$$M_i = M_{\text{ocache}} + M_{\text{ipages}} = Constant$$

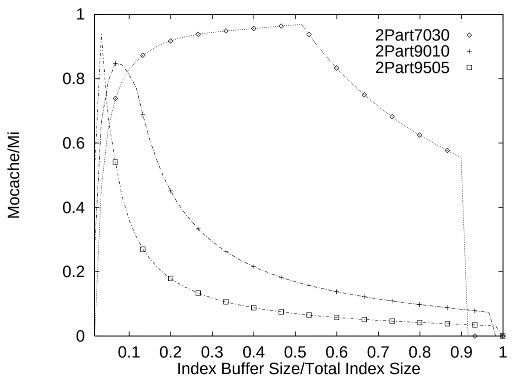
In the rest of this report, we give OD cache size as a fraction of the total index memory size. The index memory size itself is given as a fraction of the total space required for the whole index tree, i.e., when $\frac{\text{Index Buffer Size}}{\text{Total Index Size}} = 1.0$, all the pages in the index tree fits in memory.

In this study, we have chosen five access patterns, the partition sizes and access probabilities are summarized in Table 1 (note that this is the *OID access pattern*, and not the index page access pattern). We call each of these patterns a *partitioning set.* β_i^0 denotes the size of partition i, as a fraction of the total database size, and α_i^0 denotes the fraction of the accesses done to partition i.

[4] Note that memory needed for the buffering of data pages/objects is not included, this issue is orthogonal to the one studied here.

Set	β_0^0	β_1^0	β_2^0	α_0^0	α_1^0	α_2^0
3Part1	0.01	0.19	0.80	0.64	0.16	0.20
3Part2	0.001	0.199	0.80	0.64	0.16	0.20
2Part7030	0.30	0.70	-	0.70	0.30	-
2Part9010	0.10	0.90	-	0.90	0.10	-
2Part9505	0.05	0.95	-	0.95	0.05	-

Table 1: Partition sizes and partition access probabilities for the partitioning sets used in this study.



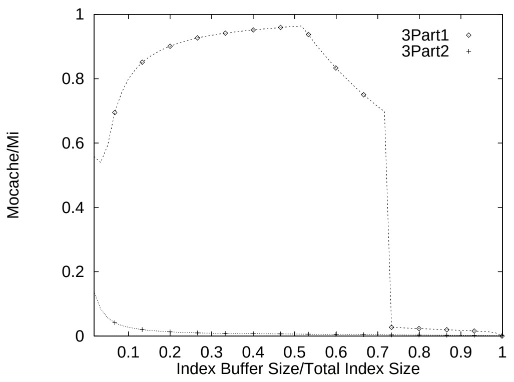


Figure 3: Optimal OD cache size with different partitioning sets.

In the two first partitioning sets, we have three partitions, extensions of the 80/20 model, but with the 20% hot spot partition further divided. In the first partitioning set, we have a 1% hot spot area, in the other, a 0.1% hot spot area. The three other sets have each two partitions, with hot spot areas of 5%, 10%, and 30%.

The analysis have been done with a database with $N_{\text{objver}} = 20$ million objects. Unless otherwise noted, results and numbers in the next sections are based on calculations with an OD size of 32 bytes, index page size of 8 KB, access pattern according to partitioning set 1, write probability $P_{write} = 0.2$, object create probability $P_{new} = 0.2$, and all objects in the database versioned. Checkpoint interval is $N_{CP} = 20000$ objects. The database size and the checkpoint interval is smaller than those used in most real world applications, but the qualitatively results and fractions stay the same when these numbers are scaled. The same applies to the number of disks. Performance can be increased by partitioning the index over N_{disks} disks. In this case, $T_b(b) = (t_r + \frac{b}{V_s}t_r)/N_{disks}$.

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. It is in situations like this that adaptive self tuning systems are most important, when parameter sets differs from the average, which systems traditionally have been tuned against.

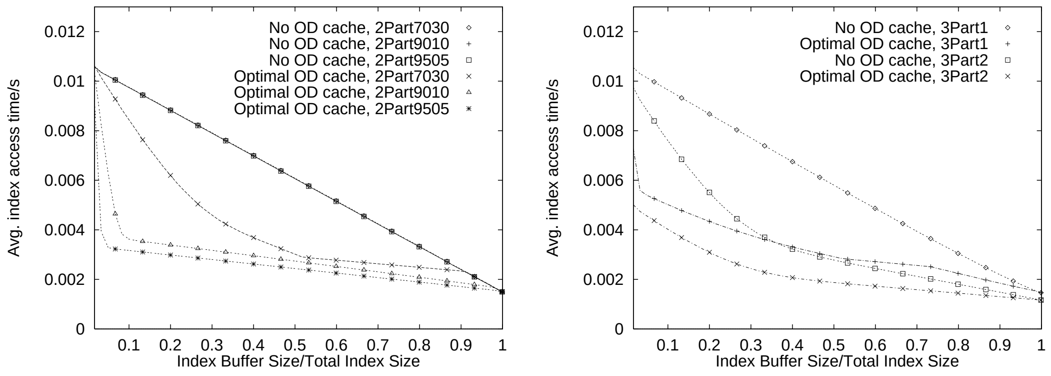


Figure 4: Access cost.

7.1 Optimal OD Cache Size

Figure 3 shows how optimal OD cache size changes with different access patterns. On the left hand side, we have the optimal OD cache size for the partitioning sets with two partitions. We see clearly how the optimal OD cache size fraction reaches the top at 5% and 10% for the partitioning sets 2Part9505 and 2Part9010, where the hot spot area is in the OD buffer. When the hot spot area gets even wider, with partitioning set 2Part7030, the hit probability for an index page when doing installation read is so small that the memory is better spent caching ODs, to reduce the lookup cost.

For partitioning sets 3Part1 and 3Part2, to the right on Figure 3, we see the same. With a significantly small and frequently accessed hot spot area, it is important to get the hottest ODs in the cache. When the cache is large enough to fit these data, the rest of the index memory is better utilized as index page buffer. With a wider hot spot area, as in 3Part1, the same phenomenon as for the 2Part7030 set happens.

7.2 Access Cost

Figure 4 illustrates how the access cost decreases with increasing amounts of available index memory. For each of the partitioning sets, the figure show the access cost without OD cache, and the access cost with an OD cache of optimal size.

From the figure, we see that even when the whole index fits in memory, the access cost is quite large. This might come as a surprise, considered that with the whole index in memory, we avoid the costly installation reads. What happens, is that a small percentage of the updates creates page splits. Even though the new pages can be written efficiently in one operation, their parent nodes have to be updated. This update is done in-place, and is a costly random write.

Interesting to note, is the access cost in the case of using the partition sets with only two partitions. In this case, if an OD cache is not employed, all accesses have to be done to the index pages. When the hottest hot spot is not small enough to make certain index pages become hot spots, the accesses to the index pages are close to uniform. As a result, the access cost is the same for all the 2 partition partitioning sets. When an OD cache is employed, we see clearly how performance increases for the access patterns with the most narrow and frequently accessed hot spots.

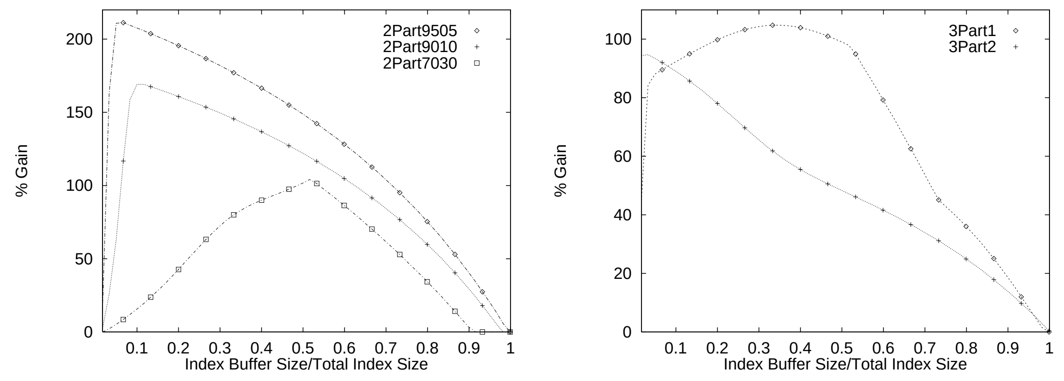


Figure 5: Gain from using optimal OD cache size, compared to no OD cache.

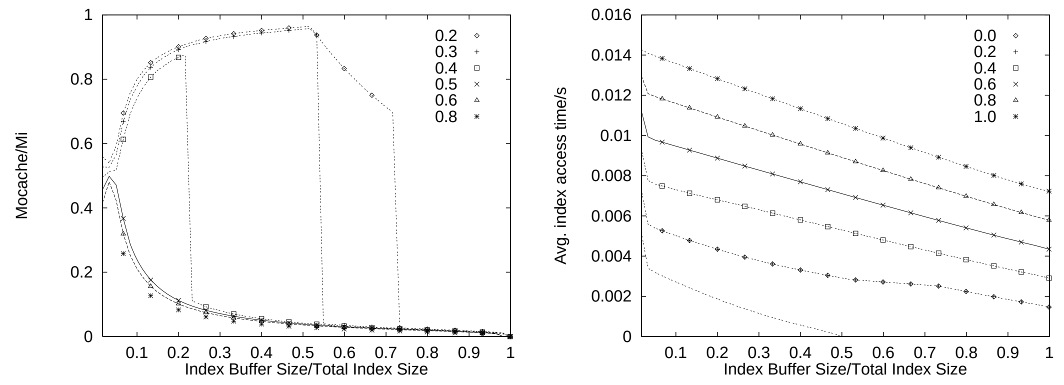


Figure 6: Optimal OD cache size with different values of P_{write} to the left, access cost with different values of P_{write} to the right.

7.3 Gain from Using OD Cache

Figure 5 illustrates the gain from employing an optimal OD cache size, compared to no OD cache at all. The gain is highest when we have a small hot spot area. The gain increases with increasing index memory size, up to the point where the whole hot spot area fits in the OD cache.

7.4 Effect of Update Ratio

The ratio of object read versus object write is important. On Figure 6 we see how changing P_{write} affects the optimal OD cache size as well as the access cost. The object create probability is held constant at $P_{new} = 0.2$.

When the update rate is relatively low, it is beneficial to use much of the memory to cache ODs. However, as the write ratio increases, the cached ODs will be less useful, because the whole index page is needed when an entry is to be updated. From the figure we see that the memory allocation strategy changes when P_{new} is between 0.4 and 0.5.

To the right on the figure, we see how the cost with different write ratios. Updating the index is costly, as is evident from the figure.

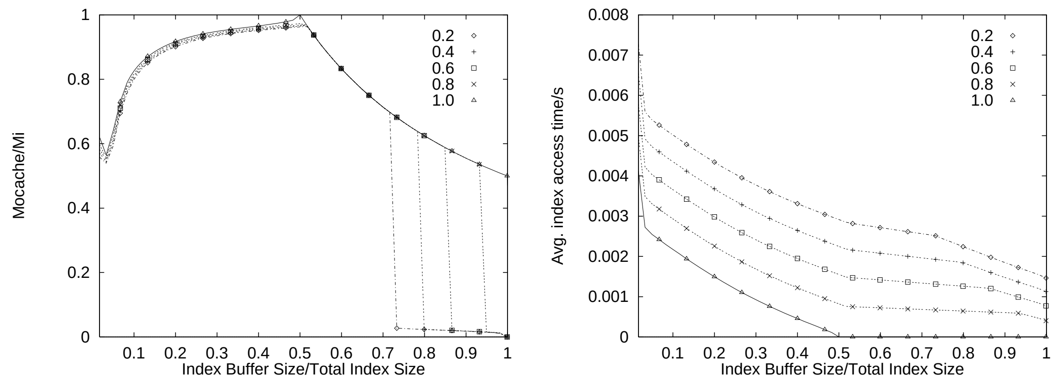


Figure 7: Optimal OD cache size with different values of P_{new} to the left, access cost with different values of P_{new} to the right.

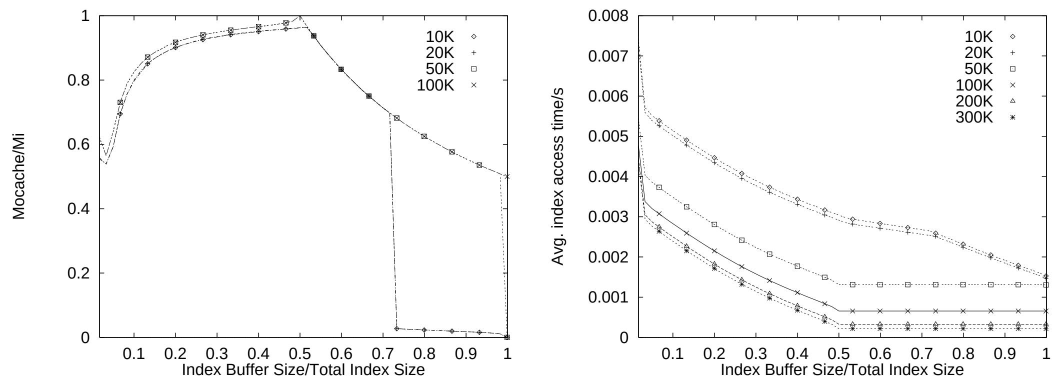


Figure 8: Optimal OD cache size with different values of N_{CP} to the left, access cost with different values of N_{CP} to the right.

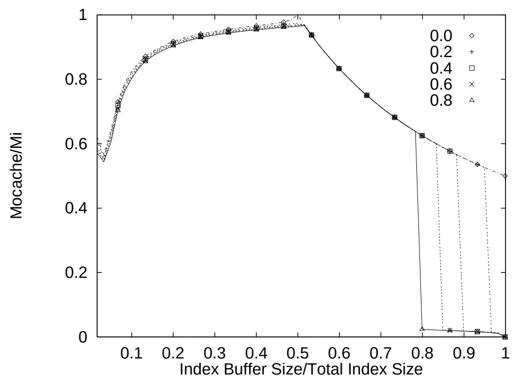
7.5 Effect of Object Creation Ratio

On Figure 7 we see how changing P_{new} affects the cost as well as the optimal OD cache size. The write probability is held constant at $P_{write} = 0.2$. The difference in optimal cache size for different values of P_{new} is only marginal, except for large memory sizes, where we have a sharp drop. The drop comes first for the low values of P_{new}. The reason is that updates needs index pages from the index as a part of the installation read. With large values of P_{new}, this is not as important.

To the left on Figure 7 we see again how important the cost of object index update is, and how the mix of create and update affects this. When objects written are mostly new objects, this can be done very efficiently. Appending ODs to the index is inexpensive.

7.6 Effect of Checkpoint Interval Length

Increasing the checkpoint interval length can reduce the access cost, illustrated to the right on Figure 8. The cost we have to pay for this, is a longer recovery time after a crash, as a larger amount of log have to be processed.



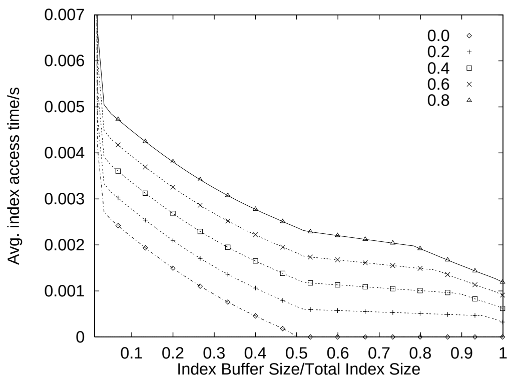


Figure 9: Optimal OD cache size with different values of $P_{versioned}$ to the left, access cost with different values of $P_{versioned}$ to the right.

We have studied how different lengths of the checkpoint interval affects the optimal M_{ocache}. Figure 8 shows the optimal OD cache size for different values for N_{CP}. An important observation is that with other parameters held constant, different values for N_{CP} give exactly one of two different values for the optimal OD cache size. This can be seen from Figure 8. Only four of the values are shown on the figure, but the result is the same for other values a well.

7.7 The Effect of Non-Versioned Objects

We have, until now, assumed that all objects in the database are versioned. Figure 9 illustrates the effects of non-versioned objects on the optimal OD cache size and the access cost. The optimal OD cache size is independent of the fraction of versioned objects.

The cost is highly dependent of the fraction of versioned objects. As emphasized earlier in this report, maintaining versions is very costly. A low fraction of versioned objects implies less index updates. The extreme is $P_{versioned} = 0.0$, which is a non-versioned database, a traditional OODB.

Note the similarities between Figure 7 and Figure 9. The reason is that, increasing the create probability and reducing the versioning, each increases the number of index appends relative to index inserts.

7.8 Page Size

The optimal page size is a compromise of two contradicting factors. Because of low locality, large page sizes in an OIDX means more wasted space in the index page buffer, and the optimal page size is thus much smaller. However, small pages also results in more levels in the tree. In our example, a page size of 16 KB gives a tree with three levels, a page size of of 2KB gives four levels, a page size of 1KB five levels, and 512B six levels. Even though in most cases upper levels of the index tree will be resident in memory, a tree with smaller page size also needs more space, reducing the buffer hit probability. We can see that there are two strategies for efficiency: Either large pager, which is particularly advantageous for the creation of objects, and small pages, to capture the fact that there is low level of sharing.

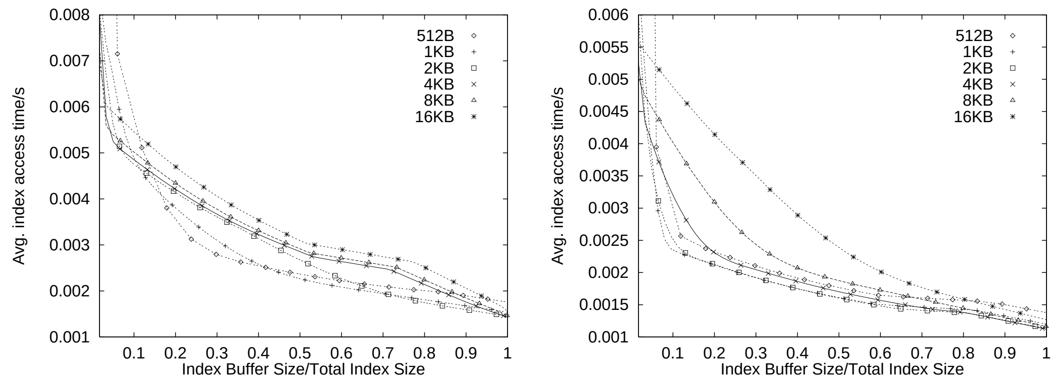


Figure 10: Access cost with different disk page sizes, 3Part1 to the left, 3Part2 to the right.

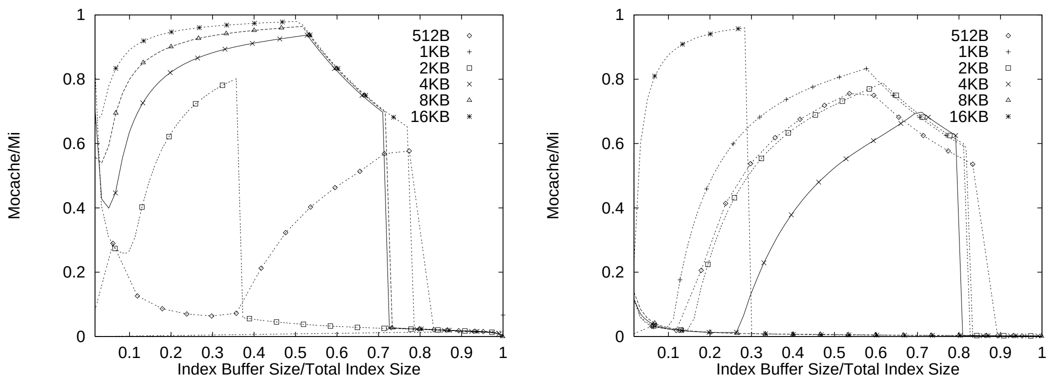


Figure 11: Optimal OD cache size with different page sizes, partitioning set 3Part1 to the left, 3Part2 to the right.

Figure 10 shows index access cost with different index page sizes. The page size giving the lowest cost over most of the area, is 2 KB. This is less than the 8 KB blocks commonly used, and this result might come as a surprise, as relational database systems have started to employ larger page sizes on their index structures. However, the index access pattern in a typical relational database systems is different from the index modeled in this report. In a relational database system, index scan is very common, which benefits from large page sizes.

Interesting to note is also how the optimal OD cache size changes with page size, illustrated on Figure 11. The reason for this, is that with large pages, we will in general benefit from more OD caching, because only a small part of the pages will be accessed. This outweights the disadvantages of the installation read needed when the page is updated.

8 Conclusions and Future Work

We have in this report developed an analytical model for OIDX access cost in a transaction time temporal database system. We have used this model to study the behavior of the OIDX under different access patterns. The results show that:

1. The OIDX access cost can be high, and can easy become a bottleneck in large TOODBs.
2. The optimal OD cache size can be large, and the gain from using an optimal size is considerable. Having an optimally tuned system is important.
3. Access pattern in a database system is very dynamic, and the system should be able to detect this, and tune the size of index page buffer and OD cache size accordingly. The cost models in this report can be of valuable use for optimizers and automatic tuning tools in temporal OODBs.

As is obvious from the results in the previous sections, OID index maintenance will be costly. To get an acceptable performance, most of the OIDX must be in memory to avoid the costly installation reads of index pages. The cost can be reduced by partitioning the index over several disks, and it is possible that the object indexing itself can be optimized, this should be studied further.

In this report, we have only studied the memory used for index pages and index entries. An issue orthogonal to this, is to find the optimal fractions of total buffer memory used for index and data pages.

The model and results in this report could also be used in the context of caching general secondary index entries in relational as well as object-oriented database systems. This issue is independent of the one studied here, but caching entries in secondary indexes should be an interesting further work.

References

[1] E. Bertino and P. Foscoli. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), 1997.

[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] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4), 1984.

[4] A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.

[5] R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B^+-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation.* The Benjamin/Cummings Publishing Company, Inc., 1993.

[6] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost model for clustered object-oriented databases. In *Proceedings of the 21st VLDB Conference*, 1995.

[7] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[8] M. E. Loomis. *Data Management and File Structures.* Prentice Hall, 1989.

[9] M. L. McAuliffe. *Storage Management Methods for Object Database Systems*. PhD thesis, University of Wisconsin-Madison, 1997.

[10] 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 (To be Published)*, 1998.

[11] K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, 1998.

[12] K. Nørvåg and K. Bratbergsengen. An analytical study of object identifier indexing. Technical Report IDI 4/98, Norwegian University of Science and Technology, 1998.

[13] T. Suzuki and H. Kitagawa. Development and performance analysis of a temporal persistent object store POST/C++. In *Proceedings of the 7th Australasian Database Conference*, 1996.

[14] J. D. Ullman. *Principles of database and knowledge-base systems*. Computer Science Press, 1988.