The Persistent Cache: Improving OID Indexing in Temporal Object-Oriented Database Systems

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

Abstract

In a temporal OODB, 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. The OIDX in a non-temporal OODB only needs to be updated when an object is created, but in a temporal OODB, *the OIDX have to be updated every time an object is updated*. We have in a previous study shown that this can be a potential bottleneck, and in this report, we present the *Persistent Cache* (PCache), a novel approach which reduces the index update and lookup costs in temporal OODBs. In this report, we develop a cost model for the PCache, and use this to show that the use of a PCache can reduce the average access cost to only a fraction of the cost when not using the PCache. Even though the primary context of this report is OID indexing in a temporal OODB, the PCache can also be applied in general secondary indexing, and can be especially beneficial for applications where updates are non clustered.

1 Introduction

In a transaction time temporal object-oriented database system (TOODB), updating an object creates a new version of the object, but the old version is still accessible. A system maintained timestamp is associated with every object version, usually the commit time of the transaction that created this version of the object.

Technical report IDI-2/99
Norwegian University of Science and Technology, Norway

An important feature of OODBs, is that an object is uniquely identified by an object identifier (OID), and that the object can be accessed via its 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 physical location of the object. In a TOODB, logical OIDs is the most reasonable alternative, because objects are frequently moved, and an OIDX is necessary anyway to index the object versions. The entries in the OIDX, which we call *object descriptors* (OD), contain administrative information, including information to do the mapping from logical OID to physical address, and the commit timestamp. The OIDX can be quite large, in non-temporal OODBs, a typical size is in the order of 20% of the size of the database itself [3]. 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 TOODBs, is that with only one version of an object (non-temporal), the OIDX needs only to be updated when an object is created. This can be done in an efficient append-only operation, and we can focus on optimizing OIDX lookups. 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 as a result index updates might become a serious bottleneck in a TOODB.

To reduce disk I/O in index operations, 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 [7]. 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. An OD cache reduces the index lookup costs considerably, and can be extended to reduce index update costs as well [11].

However, even when using a "writable" OD cache, OIDX updates are still very costly. In this report, we present an approach to further reduce the OIDX update costs. Noting that the main reason for the bottleneck against the OIDX is the low locality of entries in the OIDX tree nodes, we use an intermediate *disk resident buffer* between the main memory buffer, and the OIDX itself. We call this the *Persistent Cache* (PCache). The PCache is typically much larger than the available main memory buffer, but smaller than the OIDX itself. The entries in the PCache are managed in an LRU like way, just like a main memory cache. In addition to reducing update costs, the PCache also reduces the lookup costs.

In this report, we describe the PCache in detail, and analyze its performance by the use of cost functions. We will study optimal size of the PCache, and see how buffering of nodes in main memory should be done to optimize the PCache performance. 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 index 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 the PCache. In Section 5 we develop an the OID access cost model, and in Section 6 we use this cost model to study how different PCache sizes, memory sizes, index sizes, and access patterns affect the performance. Finally, in Section 7, we conclude the report and outline issues for further research.

2 Related Work

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[1] that have temporal OID indexing, POST/C++ [13]. The performance result presented for POST/C++ are only for relatively small databases, where the index fits in main memory, and we expect that with a larger number of objects, the OIDX would be a bottleneck.

The PCache has similarities to LHAM [8], where a hierarchy of indexes is used. One important differences between the PCache and LHAM, is that in LHAM,*all* entries in one level is regularly moved to the next, there are no LRU management, and as such, LHAM only helps in improving write efficiency, not read efficiency.

The cost models in this report are based on previous work on modeling non-temporal OODBs [10] and temporal OODBs [11], but the models have been extended to include the aspects of the PCache. The buffer and tree models have been compared with simulation results, and detailed results from the simulations with different index sizes, buffer sizes, index page fanout, and access patterns, can be found in [12].

The work described in this report is part of the Vagabond TOODB project [9].

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

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 make it possible to integrate into current OODBs if desired, and it will also be used as a basis for the OID indexing in the Vagabond TOODB.

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 [5] and LHAM [8], 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 *TIME* in the OD, and use the concatenation of OID and time, $OID \| TIME$, as the index key. By doing this, ODs for a particular OID will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the current version of a particular OID as well as retrieval of a particular time interval for an OID can be done efficiently.

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 [4]. This operation is very efficient. However, 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. However, *the OIDX is still needed*, to support navigational queries, one of the main features of OODBs compared to relational database systems. Some optimizations are possible to this OIDX, e.g., using variants of nested tree index, but as the PCache is the focus of this report, we will not elaborate more on this subject here.

3.2 Temporal Object Management

In a non-temporal (one-version) OODB, space is allocated for an object when it is created, and further updates to the object 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,[2] 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.

Not all the data in a TOODB 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 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 data management, should not reduce the performance of applications that do not utilize these features.

4 The Persistent Index Entry Cache

The ODs accessed will be almost uniformly distributed over the index leaf nodes. The OD cache makes read accesses efficient, but, in a database with many objects, most of the ODs that are updated during one checkpoint interval will reside in different leaf nodes. This low locality means that *many leaf nodes have to be updated*. When an index node is to be updated, an installation read of the node has to be done first. With a large index, the access to the nodes will be random disk accesses, and as a result, the installation read is very costly.

To reduce average access costs, the *persistent cache* (PCache) can be used. The PCache contains a subset of the entries in the OIDX, *the goal is to have the most frequently used ODs in the PCache*. In contrast to the main memory cache, the OD cache, the PCache is persistent, so that we do not have to write its entries back to the OIDX itself during each checkpoint interval. This is actually the main purpose of the PCache: to provide an intermediate storage area for persistent data, in this case, ODs.

The size of the PCache is in general larger than the size of the main memory, but smaller than the size of the OIDX. The contents of the PCache is maintained according to an LRU like mechanism. The result should be high locality on access to the PCache nodes, reducing the total number of installation reads, and making checkpoint less costly. Average OIDX lookup costs should also be less than without a PCache.

To avoid confusion, we will hereafter denote the index tree itself as the TIDX, and use OIDX to mean the combined index system, i.e., PCache and TIDX. Thus, when we say an entry is in the OIDX, it can be in the PCache, in the TIDX, or in both. This is also illustrated on Figure 1.

4.1 PCache Organization

The index related main memory buffers, the PCache, and the TIDX, are illustrated on Figure 1. The number of nodes in the PCache should be small enough to make it possible to store pointers to all the nodes in main memory. To be able to do the copying of the ODs from the PCache to the TIDX efficiently, the nodes in the PCache should be accessed in the same order as the leaf nodes in the OIDX. Therefore, the nodes in the PCache are range partitioned, each node stores a certain interval of OIDs.

Range-partitioning is vulnerable to skew, to avoid this, the partitioning can be dynamically changed (for each node, we have the OID range boundary in main memory). This is done based on the update access rates on each node. A high update access rate to a node results in a smaller interval being allocated to that node. Because the PCache nodes are frequently accessed, the repartitioning does not represent any extra cost.

It is important to note that even though reads of PCache nodes will be random disk accesses (although with small seek times), the PCache nodes will be clustered together, so that the random read will have a small seek time. There will also be several PCache node read requests at any time, so by using an elevator algorithm, the cost of reading from the PCache will be low.

4.2 PCache LRU Management

The PCache nodes are operated as an ordinary main memory cache, and when an entry is to be stored in a node in the

[2] 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.

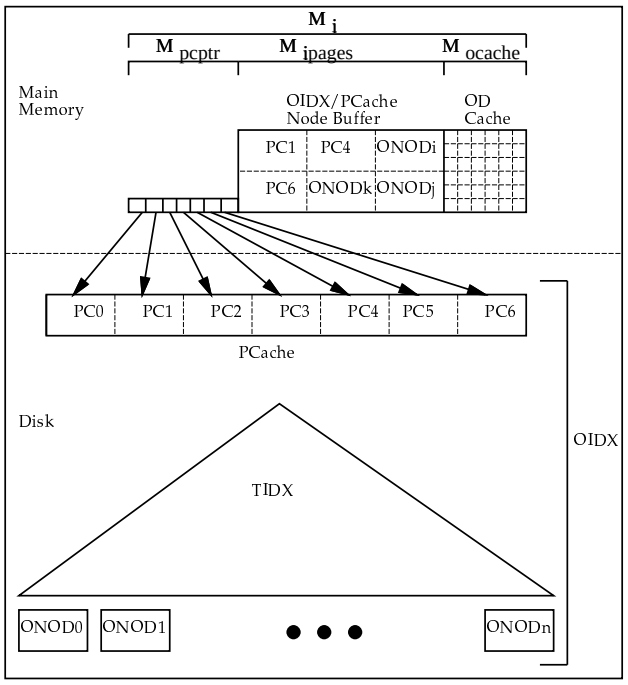

Figure 1: Overview of index and PCache. PCache nodes PC1, PC4 and PC6, and 3 OIDX nodes are in the buffer.

PCache, one of the existing entries have to be discarded.To provide access statistics, an access table is maintained for each node, with one bit for each entry in the node, and we use the clock algorithm as an LRU approximation. An access bit is set each time an entry is accessed. As for the storage of the access tables, we have several options:

1. The access table of a node could be stored *in the node itself*. A problem with this option, is that when a node is to be discarded from the main memory buffer, and entries have been accessed, the node needs to be written back to disk, even if none of the entries have been changed. This is not desirable.

2. Access tables are maintained in main memory, for each main memory resident node. When a node is discarded, i.e, due to buffer replacement, the table is discarded as well. A problem with the this option, is that until enough accesses have been done to the entries, the bit map is unreliable as a way to approximate LRU, and the "wrong" entries might get discarded.

3. Access tables are maintained in main memory for *all* PCache nodes. One problem with the this approach, is that one table for each node in the PCache is needed, but because the size of each table is small (one bit for each entry in the node), this will not represent a problem as long as the PCache is not too large. If the system crashes, the contents of the access tables will be lost, and wrong caching decisions might be done when the system is restarted. This does only affect *performance* at startup time, the ODs of committed operations are always safe on disk. To reduce the amount of wrong caching decisions at startup time, we store the access table in the node as well when the node is written to disk. Note that this differs from option 1, where the node is *always* written back when it is discarded, even when there are no updates to the ODs stored in the node.

Based on the observations above, we conclude that maintaining access tables in main memory for *all* PCache nodes is the best approach.

In addition to the access tables, each node also contains a table to keep track of the status of the entries with respect to the TIDX. One *dirty* bit is needed for each entry. The dirty bit is set each time an entry is modified or inserted into the node. Only the dirty entries need to be written back to the TIDX, entries not marked as modified can be safely discarded when needed.

4.3 Update Operations

When employing the PCache, inserting ODs resulting from object *updates* are *always* done to the entry in the PCache, never directly to the TIDX. Inserting an OD into a node, implies discarding another OD from the node, based on the LRU strategy. It is preferable to discard a non-dirty entry, so that a synchronous writeback of the dirty entry is avoided. To almost always have non-dirty slots in the nodes, dirty entries in the PCache is regularly copied over to the TIDX itself, asynchronously in the background. This is done efficiently by mostly sequential reading of the PCache nodes, and mostly sequential installation read and subsequent writing of the TIDX nodes.

4.4 Object Creations

Object creations are still applied directly to the TIDX. An OD resulting from an object creation is an efficient append operation into the TIDX, and since most created object will not be a part of the hot set, we write (append) them directly to the TIDX. In many cases, new ODs which is to be part of the hot set will be retrieved from the TIDX nodes before they are discarded from the buffer. These ODs will on access be inserted into the PCache.

4.5 Read Operations

Read operations belong to one of two classes: navigational (single object) read, and scan operations.

Single Object Read

Read operations are done by first checking if the entry to be accessed is in the OD cache or the PCache, if not found there, the TIDX itself is searched. The search in the PCache might result in one disk access if the actual node is not in buffer. When using range partitioning, there is only one candidate node, so that at most one disk access will be

needed. Accessing the TIDX can result in one or more disk accesses if the TIDX nodes are not in buffer.

When found, either in the PCache or the TIDX, the OD is inserted into the OD cache. If the OD was not already in the PCache, we have now several options, for example:

1. Insert the OD into the PCache immediately. Note that at this point, we are guaranteed to have the candidate PCache node resident in buffer, because we have probed it during the search for the OD. If we manage to get a high hit rate on the PCache, the optimal OD cache size might be quite small in this case. Note that the OD contains, in addition to the entries in the PCache, dirty entries resulted from update operations not yet installed into the PCache.

2. Insert the OD into the PCache only when it is to be discarded from the OD cache. In this way, we get good memory utilization, we do not have to use space for the OD both in the OD cache and in the PCache. However, in this case, we are not guaranteed to have the candidate PCache node resident in buffer, it might have been discarded since it was probed.

3. Never insert the OD into the PCache, only insert entries into the PCache when doing update operations. In this case, we rely on the OD cache to keep the most frequently accessed ODs, and use the PCache to be able to do efficient update of the TIDX. This strategy delays the update of the TIDX, and means that more entries can be collected before batch updating the TIDX.

The best option to choose, depends on access pattern. The possible installation read of option 2 can make it costly, and because ODs retrieved from read operations are not inserted into the PCache in the case of option 3, we only consider option 1 in this report.

Scan Operations

Scan operations must be treated different from single object read operations, as one single scan operation can make the current contents of the whole PCache to be discarded. The ODs retrieved during a scan operation will in general have less chance of being used again, it is not likely that the whole collection or container to be scanned, represents a hot set. Even if this is the case, it is possible that the number of ODs retrieved during the scan, is larger than the number of ODs that fits in the PCache. In this case, if we do a new scan over the collection/container, we will have a PCache hit probability of 0. This is similar to general buffer management in the case of scan operations.

As a result, scan operations should not update the PCache, but the PCache must be consulted during read, because recently updated ODs from the actual container/collection might reside in the PCache. However, this will not be very costly, because the contents of a physical container cached in the PCache will be clustered in the PCache's pages as well, so that the extra cost of reading the PCache pages is only marginal.

4.6 PCache-to-TIDX Writeback

The update of the TIDX, i.e., writing dirty entries in the PCache to the TIDX, will be done in the background. This is done by reading the PCache, and install the dirty entries of these nodes into the TIDX. This is done in segments, i.e., a number of nodes, and will be mostly sequential reading and writing. The PCache-to-TIDX writeback is a scan operation, and to avoid buffer pollution, nodes accessed during this operation should not affect the rest of the buffer contents, i.e., they should not make other nodes to be removed from the buffer.

The rate of the writeback is a tuning question. By giving it higher priority, i.e., doing more frequent writeback of PCache nodes, even when few entries are dirty, the probability of synchronous single entry updates resulting from PCache nodes being full of dirty entries is less likely. On the other hand, higher priority to the writeback also means that more of the disk bandwidth will be used for this purpose.

All ODs updated since the penultimate checkpoint, and still dirty in the in the OD cache, needs to be installed into the PCache or TIDX during one checkpoint period. This is not the case with the PCache-to-TIDX writeback. The period between each time the contents of a particular PCache node is written back can be very long, but still short enough to avoid overflow of dirty ODs in the PCache.

4.7 Buffer Considerations

We can have a buffer shared between TIDX nodes and PCache, as explained previously. However, this does not necessarily give optimal performance. In some cases, it might be that TIDX accesses pollutes the buffer, resulting in a low hit rate on PCache nodes. To avoid this, we can use separate buffers, one TIDX buffer, and one PCache buffer. We can also pin a certain number of the upper TIDX levels in memory, this can be advantageous because strict use of LRU is not optimal when buffering nodes of an index tree.

5 Analytical Model

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 $T_b(b) = T_{\text{start}} + T_{\text{transfer}} = t_r + \frac{b}{V_s}t_r$. Index costs can be reduced by partitioning the index over several disks. Declustering PCache nodes and TIDX nodes over several disks is straightforward.

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.

In this report, we focus on reducing access times. Obviously, the reduced access times comes at the expense of more disk space for the PCache. As disk capacity increases rapidly, with a corresponding decrease in price, we expect that in most cases, using the extra space for the PCache will be worthwhile..

As illustrated on Figure 1, a certain amount of memory, M_{ipages}, is reserved for the index page buffer, i.e, for buffering PCache and TIDX pages, 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 $N_{\text{ocache}} \approx \frac{M_{\text{ocache}}}{(S_{\text{od}}+S_{\text{oh}})}$. If we use an hash table and a clock algorithm[3] $S_{\text{oh}} \approx 8$ B.

The number of index pages that fits in the buffer is approximately $N_{\text{ibuf}} \approx \frac{M_{\text{ipages}}}{(S_{\text{P}}+S_{\text{oh}})}$.

For each PCache page on disk, we need to keep in memory the disk address to the node (4 B), the OID range boundary (4 B), and the LRU access table as described previously. This occupies a total of $M_{\text{pcptr}} = S_{PC}(\frac{S_P/S_{\text{od}}}{8}+8)$ B, where S_{PC} is the number of PCache nodes.

5.1 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 at 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 TIDX 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.

[3] A clock algorithm has performance close to LRU [2], but has less storage overhead, only one bit is needed for each entry.

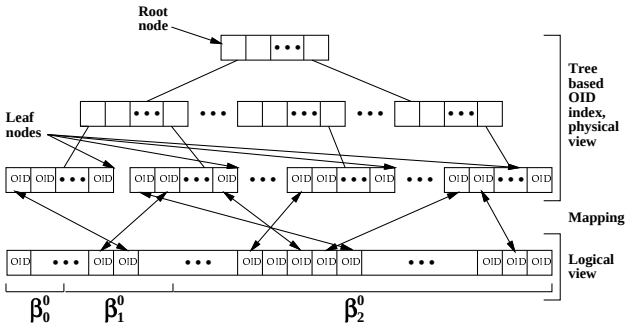

Figure 2: OID index. The lower part shows the index from a logical view, the upper part is as an index tree, which is how it is realized physically. We have indicated with arrays how the entries are distributed over the leaf nodes.

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.

5.2 Index Tree Size

If we assume an index tree with space utilization U (typically 0.69 for a B$^+$-tree), the number of leaf nodes is $N^0_{\text{tree}} \approx \frac{N_{\text{objver}}}{U\lfloor S_P/S_{\text{od}}\rfloor}$. The fanout F of internal nodes is $F = \lfloor US_P/S_{\text{ie}}\rfloor$, where S_{ie} is the size of an entry in an internal node. The number of levels in the tree is $H = 1+\lceil \log_F N^0_{\text{tree}}\rceil$. The number of nodes at each level in the tree is $N^i_{tree} = \lceil \frac{N^{i-1}_{\text{tree}}}{F}\rceil$ and the total number of nodes in the tree is $N_{\text{tree}} = \sum_{i=0}^{H-1} N^i_{\text{tree}}$.

5.3 Buffer Performance Model

Our buffer model is an extension of the Bhide, Dan and Dias LRU buffer model (BDD) [1]. Due to space constraints, we only present the most important aspects of our model in this report, but a detailed description can be found in [10].

A database in the BDD model has a size of N data granules (e.g., pages), partitioned into p partitions. 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, $\Pi_{80/20} = (0.8, 0.2, 0.2, 0.8)$. We will in the following use Π_A as short for the actual OD access partitioning set.

In the BDD model, the steady state average buffer hit probability is denoted $P_{\text{buf}}(B, N, \Pi)$, where B is the number of data granules that fits in the buffer. The buffer hit probability for data granules belonging to a particular partition p is denoted $P^p_{\text{buf}}(B, N, \Pi)$. The BDD model can also be used to calculate the total number of distinct data granules accessed after n accesses to the database, $N_{\text{distinct}}(n, N, \Pi)$.

OD Cache Hit Rate

Accesses to the OD cache can be assumed to follow the assumptions behind the BDD model, they are independent and random requests. By applying this model with object entries as data granules the probability of an OD cache hit is $P_{\text{ocache}} = P_{\text{buf}}(N_{\text{ocache}}, N_{\text{objver}}, \Pi_A)$.

General Index Buffer Model

The BDD LRU buffer model only models 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 have extended the original model to be able to analyze buffer performance in the case of hierarchical index accesses as well [12]. This is based on the 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 with H levels, we initially 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. An example of this partitioning is illustrated to the right on Figure 3, where a tree with $H = 4$ levels and $p' = 2$ leaf page partitions is partitioned into $p = 2 + (4 - 1) = 5$ partitions.

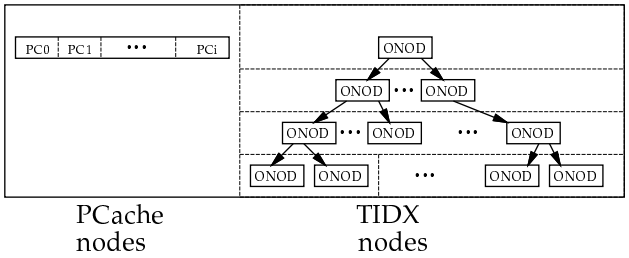

Figure 3: Access partitions.

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. As described in [12], we can use the initial OD partitioning (the OD access pattern) Π_A as basis for deriving the leaf node access partitioning Π_L .

PCache Hit Rate

We denote the probability that a certain OD is in the PCache as P_{PC}, note this actual PCache page might be in memory or on disk.

The number of ODs in each PCache node is $\lfloor \frac{S_P}{S_{\text{od}}} \rfloor$, and with a total of S_{PC} nodes, the number of ODs that fit in the PCache is $N_{PC} = S_{PC} \lfloor \frac{S_P}{S_{\text{od}}} \rfloor$. The probability that a certain OD is in one of the PCache nodes can be approximated to:

$$P_{PC} = P_{\text{buf}}(N_{PC}, N_{\text{objver}}, \Pi_A)$$

PCache and TIDX Buffer Model

When PCache and TIDX shares the same page buffer, the model have to reflect this. The access probabilities for TIDX nodes and PCache nodes are different, and as a result, in an LRU managed buffer, the hit rate will be different. In the buffer model, we use a partitioning as illustrated on Figure 3. On the figure, the PCache is one partition, and each level in the tree is a partition, with the leaf node partition further divided into two partitions, reflecting the existence of hot spot nodes (nodes belonging to one of the two leaf node partitions need not actually be physically adjacent as on the figure).

Considering the page accesses, all OIDX lookups will access one PCache page, and $(1 - P_{PC})$ of the lookups will also access the TIDX. Each OIDX lookup results on average $1 + (1 - P_{PC})H$ page accesses. Thus, the total access probability of the PCache is $\alpha_{PC} = \frac{1}{1+(1-P_{PC})H}$, which is the fraction of accessed pages that is part of the PCache partition. The TIDX access probability is $\alpha_{TIDX} = \frac{(1-P_{PC})H}{1+(1-P_{PC})H}$.

The total number of index pages is $S_{PC} + N_{tree}$. The PCache contains $\beta_{PC} = \frac{S_{PC}}{S_{PC}+N_{tree}}$ of these pages, the TIDX contains $\beta_{TIDX} = \frac{N_{tree}}{S_{PC}+N_{tree}}$ of the pages.

The TIDX partitions is further partitioned into p partitions as described above, and we denote the resulting partitioning (Figure 3) as Π_{Shared}. We denote the PCache and TIDX node buffer hit probabilities as:

$$P_{\text{buf\_PC}} = P_{\text{buf}}^{PC}(N_{\text{ibuf}}, S_{PC} + N_{\text{tree}}, \Pi_{\text{shared}})$$

$$P_{\text{buf\_TIDX}} = P_{\text{buf}}^{TIDX}(N_{\text{ibuf}}, S_{PC} + N_{\text{tree}}, \Pi_{\text{shared}})$$

As noted in Section 4.7, it can be advantageous to use separate buffers for the PCache and TIDX. In that case, $N_{\text{ibuf\_PC}}$ buffer pages are reserved for the PCache, and $N_{\text{ibuf_TIDX}}$buffer pages are reserved for the TIDX, so that $N_{\text{ibuf\_PC}} + N_{\text{ibuf\_TIDX}} = N_{\text{ibuf}}$. Denoting the tree partitioning as Π_{Tree}, the corresponding buffer hit probabilities using separate buffers are:

$$P_{\text{buf\_PC}} = \frac{N_{\text{ibuf\_PC}}}{S_{PC}}$$

$$P_{\text{buf\_TIDX}} = P_{\text{buf}}(N_{\text{ibuf\_TIDX}}, N_{tree}, \Pi_{\text{Tree}})$$

5.3.1 OIDX Lookup Cost

Assuming we have the pointers to all PCache pages in memory, and use a range partitioned PCache, at most one disk access is needed for each PCache lookup. Before a page can be read in, another have to be replaced. A page may contain dirty entries, because all read ODs from the TIDX are inserted immediately, in this case, the PCache page has to be written back. To be able to do this efficiently, we use the following strategy: On disk, we allocate space for more nodes than the number of nodes in the PCache. When we read a page, we at the same time schedule dirty page(s) for writing, to an empty slot near the node(s) to be read. In that way, the extra write cost is only marginal compared to the read. Because we at all times keep the pointers to all PCache nodes in the memory, they can be written back to different places every time.[4] We approximate the lookup cost to:

$$T_{\text{lookup\_PC}} = (1 - P_{\text{buf\_PC}})T_P$$

To access an entry in the TIDX, it is necessary to traverse the H levels from the root to a leaf page. To do this, we need $(1 - P_{\text{buf\_TIDX}})H$ disk accesses. The average cost of an TIDX lookup is:

$$T_{\text{lookup\_TIDX}} = (1 - P_{\text{buf\_TIDX}})HT_P$$

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 PCache. With a probability of P_{PC}, the entry is in the PCache. If not, the TIDX itself has to be accessed. The average cost to retrieve an entry is:

$$T_{\text{lookup}} = (1 - P_{\text{ocache}})(T_{\text{lookup\_PC}} + (1 - P_{PC})T_{\text{lookup\_TIDX}})$$

[4] It is interesting to note by using this approach, we do things according to the log-structured file system philosophy, which our Vagabond Parallel TOODB is based on [9].

5.3.2 OIDX Update Cost

We do not need to update the PCache or the index pages in the TIDX immediately after an update have been done. 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 costs during an interval, divided on the number of 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 the OD cache is large enough to keep all dirty ODs through one checkpoint interval, and that deleting and compacting pages can be done in background. This means that $N_{CP} < N_{\text{ocache}}$. Using a strategy that wrote a larger amount of ODs to the log before installing them into the TIDX, is difficult: If we did not keep all ODs not yet installed into the OIDX in the memory, we would have to search the log on each access, to check if the log contained a newer version than the one in the OIDX.

Creation of New ODs

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 (we do not distribute the entries over the old node and the new when the rightmost nodes are split), and we have clustered updates. As described previously, object creations are done directly to the TIDX, and not to the PCache. This contributes to $N_n^0 = \frac{N_{CR}}{\lfloor S_P/S_{\text{od}} \rfloor}$ 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. 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, mostly sequentially:

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

Modification of Existing ODs in the PCache

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 during one checkpoint interval is $N_U = N_{CP} - N_{CR}$.

In general, at least one OD will be inserted into each PCache page (the number of updates in one checkpoint interval is much larger than the number of PCache pages). In this case, the most efficient way to update the PCache is to read sequentially a number of PCache pages, update them,

write them back, and continue with the next segment. Assuming that PCache-to-TIDX writeback has high enough priority, we can assume that when inserting a new entry into a PCache node, there is always a non-dirty entry that can be removed. The cost of this is:

$$T_{\text{write\_to\_PC}} = 2T_b(S_{PC}S_P)$$

where S_{PC} is the number of PCache nodes.

PCache-to-TIDX Writeback Cost

The purpose of the PCache-to-TIDX writeback is to always have non-dirty slots in the PCache nodes, where new entries can be inserted. The PCache-to-TIDX writeback runs continuously in the background. The period for each round is ideally so long that each PCache node is almost full of dirty entries when it is processed.

The cost is equal to reading a number of PCache nodes (sequential reading), and writing the dirty entries back to the TIDX. If we assume each PCache node is almost full when we process it, we have fN_{PC} entries to write back in each round, where f is the PCache node dirty fill factor, i.e., the amount of dirty entries in the node. This value should ideally be close to 1, but to avoid delays in normal processing due to overflow of dirty entries in nodes, f should be sufficiently small, we will in the calculations in this report use a value of 0.90. The number of update objects during each round of PCache-to-TIDX writeback is $N_{SCP} = fN_{PC}$, which we call the *super checkpoint period*.

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 $T_{\text{lookup\_TIDX}}$ for each *distinct* object modified. The number of distinct updated objects is:

$$N_{DU} = N_{\text{distinct}}(N_{SCP}, N_{\text{objver}}, \Pi_A)$$

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

$$N^V_{DU} = P_{\text{temporal}}N_{DU}$$

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

$$N_m = N_{\text{distinct}}(N^V_{DU}, N^0_{tree}, \Pi_L)$$

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 [6], 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 $P_{\text{split}} \approx \frac{1}{\lceil (S_P/S_{\text{od}})/2 \rceil - 1}$. 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:

$$\begin{aligned} T'_{\text{writeback}} &= (N_m + N_m 2P_{\text{split}})T_P \\ &= N_m(1 + 2P_{\text{split}})T_P \end{aligned}$$

The equation above assumes that there will on average be less than one entry to be inserted in each leaf node. That is the case as long as we use the following optimization: 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:

$$T''_{\text{writeback}} = 2T_b(N_{\text{tree}}S_P)$$

giving:

$$T_{\text{writeback}} = \min(N_m T_{\text{lookup\_}TIDX} + T'_{\text{writeback}}, T''_{\text{writeback}})$$

Average Index Update Cost

The average index update cost per object is the total cost of updating the PCache and the PCache-to-TIDX writeback, divided on the number of updated objects:

$$T_{\text{update}} = \frac{T_{\text{writenew}}}{N_{CP}} + \frac{T_{\text{write\_to\_PC}}}{N_{CP}} + \frac{T_{\text{writeback}}}{N_{SCP}}$$

Note that the total PCache-to-TIDX writeback is for one super checkpoint period, while the PCache update and object creating cost is per ordinary checkpoint period.

5.4 OIDX Access Cost Without PCache

In a system with no PCache, the OIDX lookup cost is:

$$T_{\text{lookup}} = (1 - P_{\text{ocache}})T_{\text{lookup\_TIDX}}$$

Without a PCache, we need to write back all ODs to the TIDX each checkpoint interval. This is similar to PCache-to-TIDX writeback, except that we now write back only $N_{CP} - N_{CR}$ entries instead of N_{SCP}, which makes it more difficult to do it efficiently. The average update cost is:

Set	β_0^0	β_1^0	β_2^0	α_0^0	α_1^0	α_2^0
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
2P8020	0.20	0.80	-	0.80	0.20	-
2P9505	0.05	0.95	-	0.95	0.05	-

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

$$T_{\text{update}} = \frac{T_{\text{writenew}}}{N_{CP}} + \frac{T_{\text{writeback\_TIDX}}}{N_{CP}}$$

where $T_{\text{writeback\_TIDX}}$ is calculated according to the equations used for $T_{\text{writeback}}$, except that $N_{CP} - N_{CR}$ is used instead of N_{SCP} in calculating $N_D U$. When calculating the buffer hit probabilities, the index page buffer is only used for the TIDX, with the absence of a PCache, no memory is used for PCache pointers and tables.

6 Performance Study

We have now derived the cost functions necessary to calculate the average cost of OIDX access under different system parameters and access patterns, with and without the use of a PCache. We will in this section study how different values for these parameters affects the access cost, which conditions using a PCache is beneficial, and optimal sizes for the PCache. 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 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:

$$T_{\text{access}} = (1 - P_{\text{write}})T_{\text{lookup}} + P_{\text{write}}T_{\text{update}}$$

Our goal here is to minimize T_{access}. We measure the gain from using a PCache, with optimal parameter values, as:

$$\text{Gain} = 100\left(\frac{(T_{\text{access\_noPCache}} - T_{\text{access}})}{T_{\text{access}}}\right)$$

where $T_{\text{access\_noPCache}}$ is the access cost when not using a PCache. In the rest of this report, we give PCache size as a fraction of the TIDX size.

It is difficult to know what kind of access pattern that will be experienced in TOODBs. It is possible to do predictions based on current access patterns, but we believe that it is quite possible that when support for temporal features become common, application developers can utilize these in new ways. The access patterns used in this report does not necessarily represent any of these, but we will use them to show that the gain from using the PCache is considerable, under most conditions and access patterns.

We have used four access patterns. The partition sizes and access probabilities are summarized in Table 1 (note that this is the *OID* access pattern Π_A, and not the *index page* access pattern Π_L). In the first partitioning set, we have three partitions, extensions 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 TOODBs, with a large cold set, consisting of old versions. The two other sets in this analysis have each two partitions, with hot spot areas of 5% and 20%.

Parameter	Value	Parameter	Value
M_{ocache}	0.1 M	N_{objver}	100 mill.
V_s	50 KB	U	0.67
t_r	8.33 ms	N_{CP}	$0.9N_{\text{ocache}}$
S_P	8 KB	P_{new}	0.2
S_{od}	32 B	P_{write}	0.2
S_{oh}	8 B	P_{temporal}	0.8
S_{ie}	16 B		

Table 2: Default parameters.

Unless otherwise noted, results and numbers in the next sections are based on calculations using default parameters as summarized in Table 2.[5] Note that in this report, when we talk about available main memory, we only consider the memory available for index related buffering, M_i. Main memory for object page buffering is orthogonal to this issue.

With the values in Table 2, the ODs would occupy ≈ 3.1 GB if stored compactly. Typically, the objects themselves occupies at least four times as much space as the OIDX, if this is reflected in available main memory buffer, $M_i = 50$ MB should imply a total buffer memory of 2-300 MB. In this study, we mainly investigate the index memory interval from $M_i = 1$ MB to $M_i = 50$ MB, as this is the most dynamic area of OIDX access cost, but we will also show how the availability of larger amounts of memory affects performance.

6.1 The Effect of Using a PCache

Figure 4 illustrates the typical cost involved in OIDX access, using the default parameters, but with different index memory sizes M_i. The gain is from 50% to several 100%. We see that the PCache is especially beneficial with relative small index memory sizes compared to the total index size. As the index size increases, the gain decreases (but as we will show in Section 6.4, the gain actually increases again with larger main memory sizes).

[5] 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 would be interesting, when parameter sets differs from the average, which systems traditionally have been tuned against.

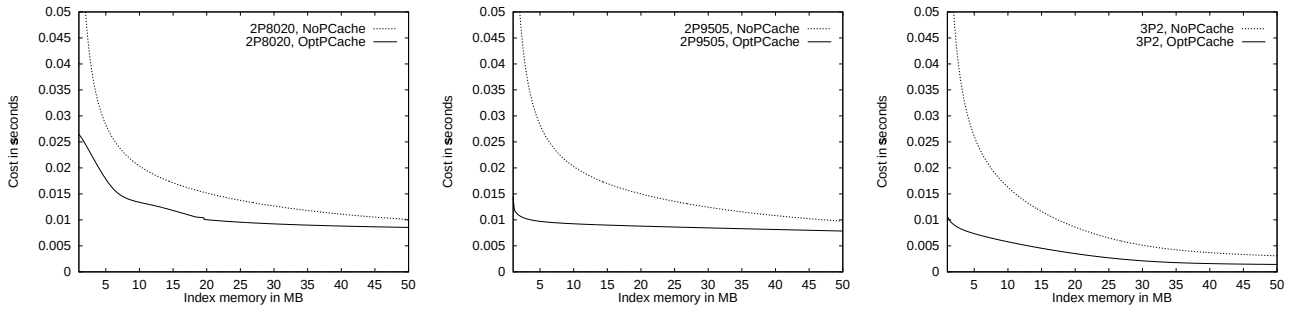

Figure 4: OIDX access cost with and without employing a PCache, with access patterns according to 2P8020, 2P9505, and 3P3.

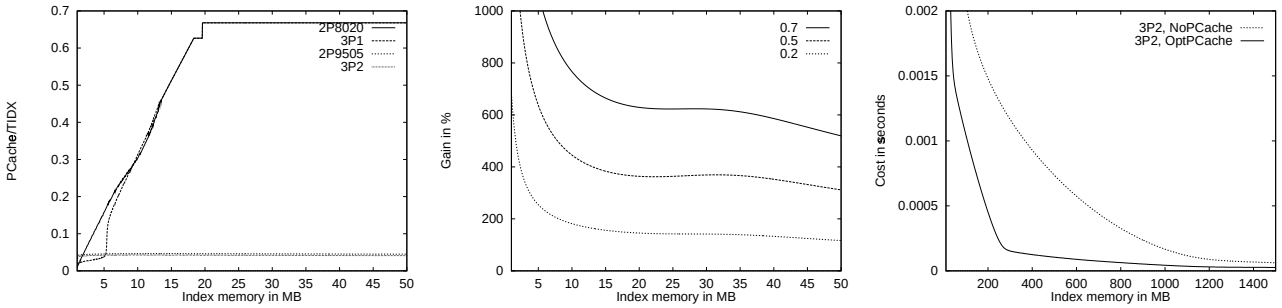

Figure 5: Optimal PCache size for different access patterns to the left. In the middle, the effect of different update ratios P_{write} for access pattern 3P2. To the right, we have the OIDX access cost with and without employing a PCache, employing large index memory sizes.

6.2 Optimal PCache Size

The optimal PCache sizes are illustrated to the left on Figure 5. With access pattern 2P8020 and 3P1, most of the available index memory, except the memory reserved for the OD cache, is used to store the pointers and LRU tables for the PCache. 2P9505 and 3P2 have more emphasized hot spot ares, in this case, a smaller PCache is optimal, just enough to store the hot spot area.

6.3 The Effect of Different Update Ratios

The main purpose of the PCache is to increase OIDX update performance. This is illustrated very well on the middle subfigure of Figure 5, which shows the gain using different values for P_{write}.

6.4 The Effect of Larger Amounts of Memory

The right subfigure of Figure 5 illustrates that the gain from using a PCache is also valid when large main memory buffer is available. The minimum gain here is when $M_i \approx 80$ MB. At that point, the gain is 94%. It then increases again, until $M_i \approx 300$ MB, where the gain is 666%. After that, the gain from using PCache slowly decreases, with increasing amounts of available main memory.

6.5 The Effect of Different Page Sizes

The page size is an important factor in determining the indexing performance. 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 a higher tree. 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. We have studied optimal page sizes for the different access patterns, with possible page sizes between 2 KB and 64 KB. All shows that a small page size is beneficial. 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.

6.6 PCache Using Separate Buffers

We have also done the analysis with separate buffers for the PCache and TIDX. The analysis shows that a cost reduction of a few percent, typically from 2 to 3%, can be found. However, in this case, it is very important with accurate buffer partitioning. This assumes knowledge of current access pattern at all times, something which is difficult in practice. LRU buffer management is in this sense self adaptive, and with only marginal improvement when using separate buffer, we advice against using separate buffers.

7 Conclusions and Future Work

We have in this report described the PCache, and how it can be used to improve performance in an TOODB. We have developed cost models which we have used to analyze the improved performance and characteristics of the PCache, under different access patterns, and memory and index sizes. The results show that:

1. The OID indexing cost in a TOODB will be large, but can be reduced by the use of a PCache.
2. The gain from using a PCache can be large.
3. The gain is especially good when using an *optimal* size of the PCache. Having an optimally tuned system is important. Access pattern in a database system is 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 TOODBs.

In the report, we have described several strategies for PCache LRU table storage, and PCache update strategies when doing read operations. These issues are interesting further work. It is possible that by combining several strategies in a dynamic adaptive PCache can improve performance even more, and making the system less vulnerable to rapidly changing access patterns and variants of data skew.

This report described the PCache used to improve OID index in TOODBs. The PCache should also be applicable to general secondary indexing, especially interesting is applications where updates are not clustered, i.e., have low locality.

Acknowledgments

We would like to thank Kjell Bratbergsengen for valuable comments and ideas, which have significantly improved the work presented in this report.

References

[1] 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.

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

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

[4] 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.

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

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

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

[8] 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.

[9] 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.

[10] 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, DEXA'98*, 1998.

[11] 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.

[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. Available from http://www.idi.ntnu.no/grupper/DB-grp/.

[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.