An Analytical Study of Object Identifier Indexing

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

The object identifier index of an object-oriented database system is typically 20% of the size of the database itself, and for large databases, only a small part of the index fits in main memory. To avoid index retrievals becoming a bottleneck, efficient buffering strategies are needed to minimize the number of disk accesses. In this report, we develop analytical cost models which we use to find optimal sizes of index page buffer and index entry cache, for different memory sizes, index sizes, and access patterns. Because existing buffer hit estimation models are not applicable for index page buffering in the case of tree based indexes, we have also developed an analytical model for index page buffer performance. The cost gain from using the results in this report is typically in the order of 200-300%. Thus, the results should be of valuable use in optimizers and tools for configuration and tuning of object-oriented database systems.

1 Introduction

In a large OODB with logical object identifiers (OIDs), the OID index (OIDX) can be quite large, typical in the order of 20% of the size of the database itself [8]. 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 is not applied. In this report, we will study OID index retrieval analytically, and use the results to reduce the average number of disk accesses needed to retrieve an index entry. This can reduce the cost of OID mapping considerably, in many cases to only a fraction of the original cost.

Traditionally, the most recently used *index pages* have been kept in the buffer pool to make OID mapping efficient, usually managed as an LRU chain. However, if the index entries have low locality, which is often the case for OID indexes, only a little part of the information in the pages in the buffer is really of interest. To better utilize the memory, it is possible to keep the most recently used *index entries* in an OID entry cache (OE cache), as is done in the Shore OODB [12]. This can improve performance considerably, but it is important to know how much of the memory should be used for buffering index pages, and how much should be used for caching[1] index entries. This issue has not previously been studied in detail. The OIDX access cost is also an issue that has been underestimated in research literature, the OID mapping cost have not been included into the cost models. The average number of disk accesses (based on equations that will be derived later in this report) for various index

[1]In this report, we use buffer and buffering when we talk about index *pages* in memory, and cache and caching when we talk about index *entries* in memory. This is established terminology, but apart from the naming, there is no real difference between buffering and caching techniques in this case.

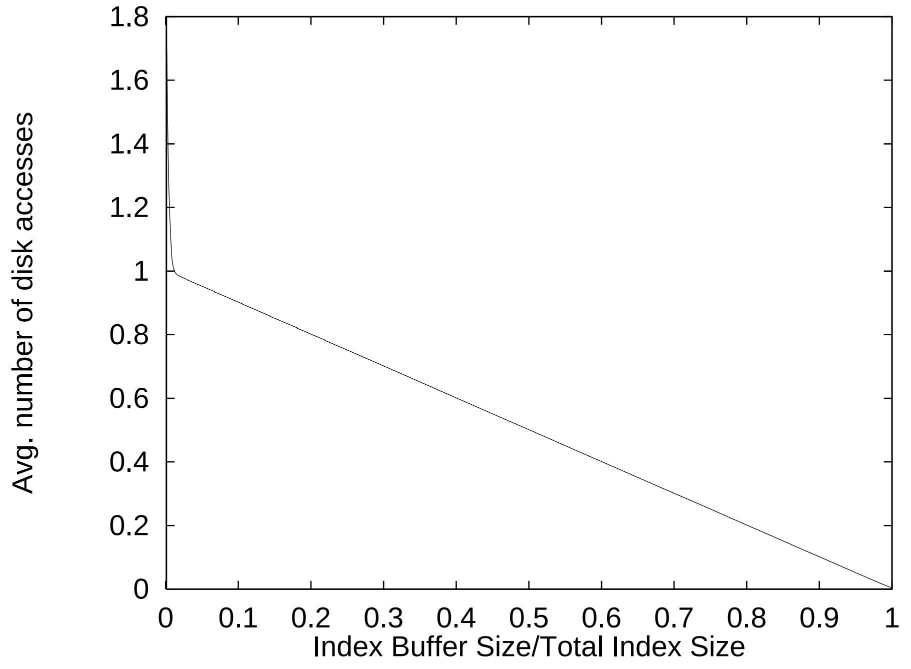


Figure 1: Average OID mapping cost.

memory sizes is illustrated in Figure 1. It is evident from the figure that this cost can not be ignored, and we will later show that there is much to be gained by optimization.

As the amount of memory increases, most of the frequently accessed part of the index and database will fit in main memory, and in this case, it is the cost of the *infrequently accessed* data and index structures that will represent the disk bottleneck. In this report, we will develop a cost model for OID mapping, which can be incorporated into existing models. We will also study how index buffer memory should be partitioned between *index page buffer* and *index entry cache*, and describe how different memory and database sizes affects this partitioning.

The organization of the rest of the report is as follows. Section 2 gives an overview of related work. In Section 3 we explain why we use a modeling approach. In Section 4 we describe some special characteristics of OID indexing, together with our assumptions regarding access pattern. In Section 5 we present the Bhide, Dan and Dias LRU buffer model (BDD model), which our index buffer model in Section 6 is based on. In Section 8 we develop an OID retrieval cost model, and in Section 9 we study how different memory and index sizes affects the performance. Finally, in Section 10, we conclude the report and indicate topics for future work.

2 Related Work

There have been several approaches to estimate the number of page accesses in the case of non-hierarchical files [3, 14, 13], and studies of the characteristics and validity of these estimates [10, 6]. However, they have not taken access pattern and buffering into account. Accesses have been assumed to be uniform, and an infinite buffer has been assumed, where only the first access to a page is counted.

Recently, other approximations have been developed, the most interesting is the work done by Bhide, Dan and Dias [2], which is an improvement of a model presented by Dan and Towsley [5].

They model requests through an LRU buffer, and the model is valid under the assumption that each request is independent of all previous requests. We will describe this model in more detail in Section 5.

Modeling buffer in the case of hierarchical files, where traversing the tree makes the request independency assumption invalid, complicates the situation. One modeling approach is presented in [11], where an I/O model for index scans with LRU buffer is presented. However, this model assumes a complete scan of the index, and is thus a model for uniform access to the index, all entries have the same probability of access (essentially one access during one scan).

Several cost models for OODBs have been developed, for example [9, 1], but they do not include buffer and OID mapping cost.

3 The Benefits of Analytical Modeling

To be able to compare different algorithmic and architectural options, simulation and analytical modeling have been used extensively in database related research. For evaluations of complex systems, simulations has been most commonly used, mainly because of its proximity to implementation and the problem of modeling concurrent events. However, simulations have some very important shortcomings: 1) they are *time consuming*, and as a result only a small part of the parameter space can be explored, and 2) *it is not always easy to explain the results*! By having the opportunity to easily change parameters, and get the results fast, it is easier to determine dependencies with analytical models. Thus, analytical modeling should be used when possible.

Analytical modeling in database research has mostly focused on I/O costs. This is done under the assumption that I/O is the bottleneck. 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). Another 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.

4 Object Identifier Indexing

An object in an object-oriented database system (OODB) is identified by a unique *object identifier* (OID), which can be *physical*, or *logical*. If the former is used, the disk page where the object resides is given directly from the OID, if the latter is used, it is necessary to use an index to find the page where the object resides. Even though a physical OID has a potential performance benefit, it also has some major drawbacks: relocation and migration of objects is difficult, which also make schema change more difficult. Therefore, logical OIDs are generally the preferred choice. An OIDX is usually realized as a hash file or a B-tree. Based on simulation results from a comparison study by Eickler et al. [8], and our own study of OID indexing, we believe B-trees or ISAM variants to be the best suited for OID indexing, and therefore we limit this discussion to tree based indexing. It should be noted, however, that much of this report is still relevant to hash file indexing, the only difference is the index access equations, which are simpler for hash files.

The number of OIDs can be very large, and a fast and efficient index structure is needed. An OIDX has two special properties that are exploited to achieve this goal:

- The keys in the index, the OIDs, are not uniformly distributed over a domain as keys commonly are supposed to be. The unique part of an OID is usually an integer that will always be assigned

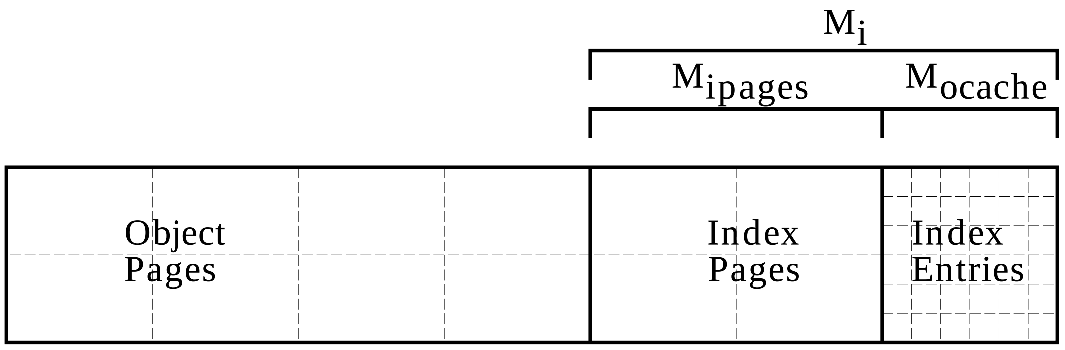


Figure 2: Buffer in a page server OODB.

monotonic increasing values. As a result, there will never be insertions of new key (OID) values between existing keys (OIDs).

- If an object is deleted, the OID will never be reused.

In general, it is difficult to get the space utilization in a B-tree better than 0.69. When indexing OIDs, however, it is possible to get a space utilization close to 1.0 by combining the knowledge of the OIDX properties and using techniques as tuned splitting. A more unfortunate result of the above properties, is that in general, it is difficult to get an index with good locality in pages. Thus, clustering can not be exploited.

Getting better locality on index pages would require a different OID generation scheme, for example, to leave sufficient space between each OID so that new OIDs could be inserted near an old OID if clustering is desired (and beneficial). The result could be more efficient OID entry search. However, this would require keeping an entry in the index for deleted objects as well, resulting in a larger index. Another problem with this approach, is that at least one extra index access is needed each time a new object is created, to check if an OID already exists. To our knowledge, this approach is not used in any existing system, and it is not clear whether it would be beneficial to do it this way.

4.1 Buffer

The buffer in a page server OODB is used for buffering object pages, index pages, and optionally, index entries, as illustrated on Figure 2. In the rest of this report, we will denote the index page buffer size as M_{ipages}, the OE cache size as M_{ocache}, and the sum of these as M_i, as illustrated.

In a real implementation, there will usually not be separate buffer areas for object and index pages. All index and data pages resides in a common buffer, managed by some buffer management policy, for example DBMIN [4]. With this buffer management policy, it is possible to put restrictions on how much of the buffer a certain task/transaction can allocate. This is necessary to avoid situations where some operation do something that replaces all the pages in the buffer, e.g. a scan operation. With such a management strategy, it is also possible to control the amount of memory used for indexing, and even without such policies, it is still easy to know the amount of buffer occupied by index pages by simple bookkeeping.

The size of the OE cache can be fixed (set at system startup time), or adaptive. We will later in this report study how an adaptive OE cache can improve performance over a fixed size OE cache.

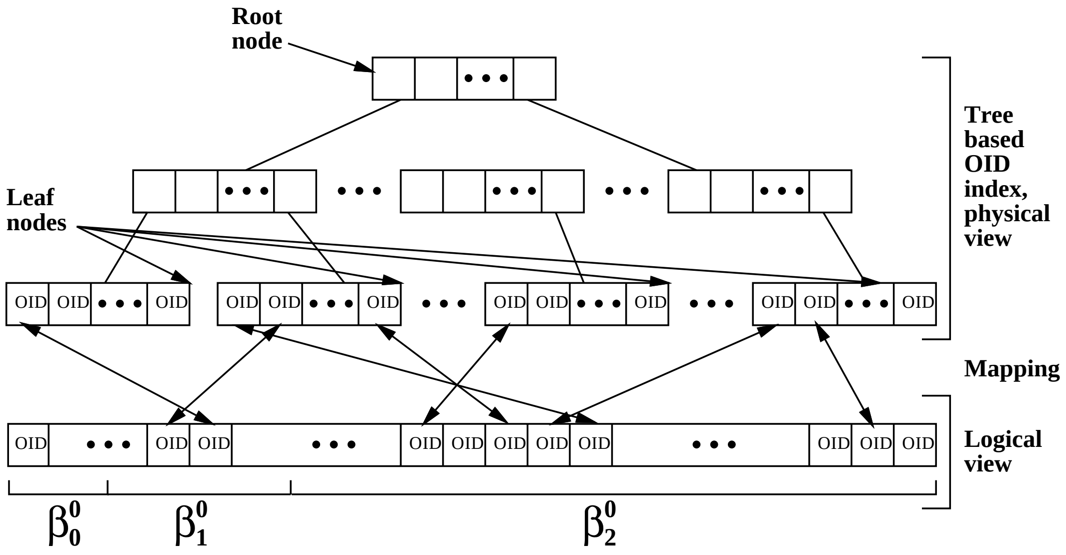


Figure 3: OID index. The lower part shows the index from a logical view, the upper part is as an index tree, which is how it is realized physically. We have indicated with arrays how the entries are distributed over the leaf nodes.

Partition Size, Set 1	Partition Size, Set 2	Partition Access Probability
$\beta_0^0 = 0.01$	$\beta_0^0 = 0.001$	$\alpha_0^0 = 0.64$
$\beta_1^0 = 0.19$	$\beta_1^0 = 0.199$	$\alpha_1^0 = 0.16$
$\beta_2^0 = 0.80$	$\beta_2^0 = 0.80$	$\alpha_2^0 = 0.20$

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

4.2 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 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 3. We consider a database in a stable condition, with N_{obj} objects (and hence, N_{obj} index entries).

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 has to be reflected in the model.

In the simulations and analysis described in this report, we have employed two different partition sets, each with three partitions (note that this is accesses to OIDs, and not to pages in the index). In both partitioning sets, the 20% hot spot area from the 80/20 model is partitioned. In the first partitioning set, we have a 1% hot spot area, in the other, a 0.1% hot spot area. The partition sizes and partition access probabilities are summarized in Table 1. β_i^0 denotes the size of partition i, as a

Partition Fraction	Partition Size (Words)			# of Words in the Book			Access Probability		
	DB	EG	FS	DB	EG	FS	DB	EG	FS
$\beta_0^0 = 0.01$	241	25	94	432188	16341	87174	0.66	0.42	0.50
$\beta_1^0 = 0.19$	4583	465	1774	178910	17744	64640	0.27	0.46	0.37
$\beta_2^0 = 0.80$	19294	1960	7470	44830	4435	23531	0.07	0.12	0.13

Table 2: Partition sizes and partition access probabilities for three analyzed books, which shows that our assumptions regarding access pattern can be justified.

fraction of the total database size, and α_i^0 denotes the fraction of accesses done to partition i.

Some people might question the validity of the assumptions above, and wonder whether such a large amount of requests going to a small area is a realistic assumption. To show that this is quite reasonable, we will give an example. Imagine a spell checker, which spell checks a document by comparing each word in the document against the words in a dictionary, accessed by an index. As an example, we have used three books: a Danish bible (DB), an English version of the Genesis (EG), and a Norwegian file systems book (FS). If we think of the distinct words in each of the books as dictionaries (24118, 2450 and 9338 unique words, respectively), and do a spell check of the books (655928, 38520, and 175345 word accesses), we get the access probabilities for the 1%/19%/80% partitions as summarized in Table 2. They show that our assumptions regarding access probabilities can be justified.

4.3 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 is not clustered in the index. This is illustrated in Figure 3, 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 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, $\beta_0^0 N_{obj} < N_{\text{tree}}^0$, and 2) the number of hot spot entries is larger than the number of leaf nodes, $\beta_0^0 N_{obj} > N_{\text{tree}}^0$.

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 an hot spot entry, and those which have not:

$$\begin{aligned}\beta_{L0} = &\ \beta_0^0 N_{obj}/N_{\text{tree}}^0 \\ \beta_{L1} = &\ 1 - \beta_{L0}\end{aligned}$$

with access probabilities:

$$\begin{aligned}\alpha_{L0} = &\ \alpha_0^0 + \beta_{L0} \sum_{i=1}^{p} \alpha_i^0 \\ \alpha_{L1} = &\ 1 - \alpha_{L0}\end{aligned}$$

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:

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

with access probabilities:

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

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

5 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 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 is assumed to be uniform. All accesses are assumed to be independent.

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:

$$B_i(n) = \beta_i N(1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n})$$

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:

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

and the overall buffer hit probability is:

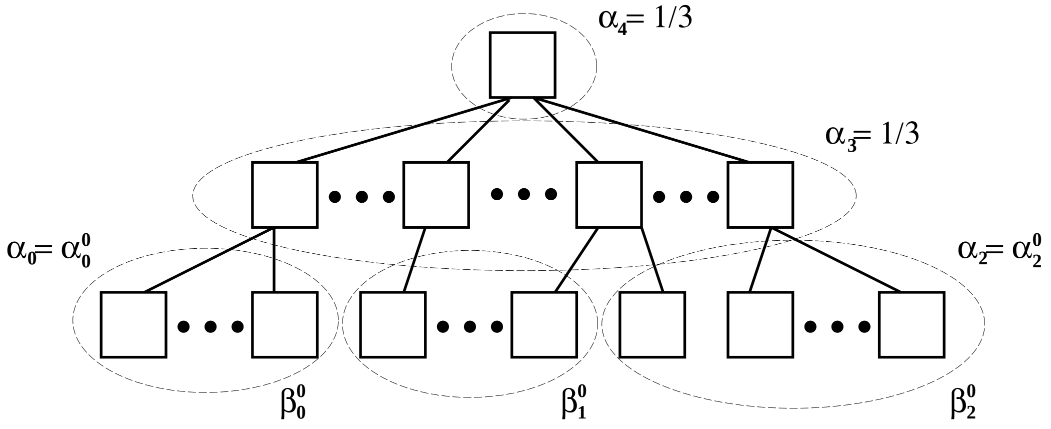


Figure 4: An index tree and its partitions.

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

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

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

6 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}. This give a fanout $F = \lfloor S_P / S_{\text{ie}} \rfloor$ and with a database consisting of N_{obj} objects to be indexed, the number of leaf nodes is:

$$N_{\text{tree}}^{0} \approx \frac{N_{\text{obj}}}{\lfloor S_P / S_{\text{ie}} \rfloor}$$

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:[2]

$$H = 1 + \lceil \log_F N_{\text{tree}}^{0} \rceil \tag{2}$$

[2]Note that this equation has been corrected from the original version of this technical report.

We initially have H partitions. Each of these partitions is 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 makes 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:

$$\alpha_i = \begin{cases} \frac{\alpha_{Li}}{H} & \text{if } i < p' \\ \frac{1}{H} & \text{if } p' \leq i < p \end{cases}$$

and

$$\beta_i = \begin{cases} \frac{\beta_{Li} N^{0}_{\text{tree}}}{N_{\text{tree}}} & \text{if } i < p' \\ \frac{N^{i}_{\text{tree}}}{N_{\text{tree}}} & \text{if } p' \leq i < p \end{cases}$$

This partitioning is illustrated on Figure 4, where we have indicated the partitioning of the leaf level, and marked each partition with an ellipse and the access probability α_i of the respective partition. We denote the number of leaf nodes as N^{0}_{tree}. The total number of nodes in the tree, N_{tree}, can then be calculated as:

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

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:

$$P_{\text{buf\_ipage}}(B, N_{\text{tree}}) \tag{3}$$

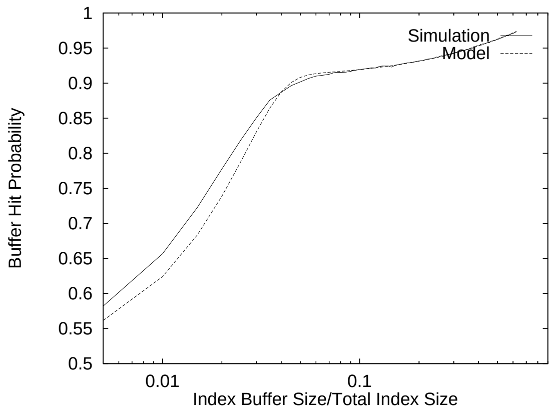
where $B = \lfloor \frac{M_{\text{ipages}}}{S_P} \rfloor$ is the buffer size, and N_{tree} is the total number of index pages in the tree.

In the analysis above, we have assumed that a complete traversal is needed from root to leaf for every leaf node access, a *traverse* strategy. In most index implementations, however, the leaf page would be self describing. When doing a search in the tree, we would first check if the relevant leaf page is already in the buffer. Only if the leaf page is not resident, we would need to traverse the tree. We call this a *no traverse strategy*. The choice of strategy has little impact on the results from the buffer hit equations, which we will show when we compare the model and simulation results in the next section.

7 Validation of the Index Buffer Model

To validate the analytical model, we have compared it with simulation results. The simulations have been performed with different index sizes, buffer sizes, index page fanouts, and access patterns.

The simulator itself is quite simple. It maintains an LRU chain of index pages currently resident in the buffer. The simulator can operate with either the traverse or the no traverse strategy (cf. Section 6).



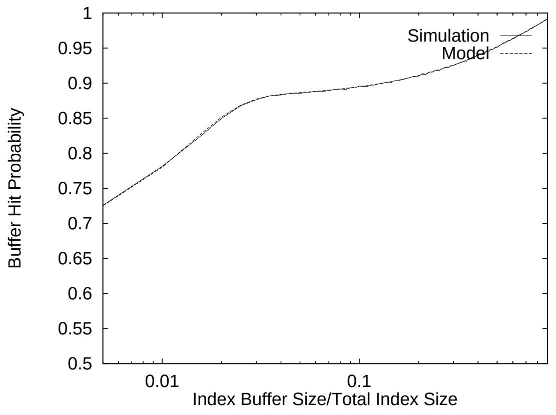


Figure 5: Overall buffer hit probability with different buffer sizes, fanout F=64 to the left, and fanout F=2048 to the right.

With the no traverse strategy, we are satisfied if the leaf page requested is in the buffer. With the traverse strategy, we assume that the leaf pages in the buffer do not contain enough information to be accessed directly. In this case, we traverse the tree from the root node to leaf page, and do a buffer allocation and replacement each time we ask for a node that is not resident in the buffer. An access to a node resident in buffer, makes the node move to the front of the buffer chain. The equations derived in the previous section, assumed a hot buffer. Therefore, we warm up the buffer by doing a large number of requests, before we start measuring the buffer hit probability.

All simulation results in this section are results from simulations with an index tree with 200000 leaf pages.[3] We have also performed simulations with larger index trees, but with the same qualitative result.

Figure 5 shows the overall buffer hit probability for an index with fanout F=64 and fanout F=2048 (note that the number of leaf pages is the same for both indexes). We see clearly how the buffer hit performance increases close to the point where a new complete index tree level have space in the buffer. After that point, it rises more slowly, until it reaches the point when most of the next level fits completely in the buffer. On the figure, we also see how close the model is to the simulation results (we have used a logarithmic scale for the buffer size axis to emphasize deviations, without logarithmic scale, the curves would be overlapping).

Figure 6 shows the relative deviation between estimated and simulated buffer hit rate. The deviation is very small. The left hand plot is with partitioning set 1, the right hand plot is with partitioning set 2 (cf. Table 1).

In the previous section, we explained how traversal from root to leaf can be avoided when the desired leaf node was resident in buffer. To see if this affects the buffer hit probability, we have compared the simulations done with the no traverse strategy, and compared the results with the results of the analytical model, which assumed complete traversal. In Figure 7, we have plotted the relative deviation between estimated and simulated buffer hit rate in the case of a no traverse strategy. This shows that the deviation is small in this case as well. We also measured the deviation with other partitioning sets, and got the same result.

[3]200000 leaf pages are, with a leaf page size of 1024 entries, sufficient to index approximate 200 million objects.

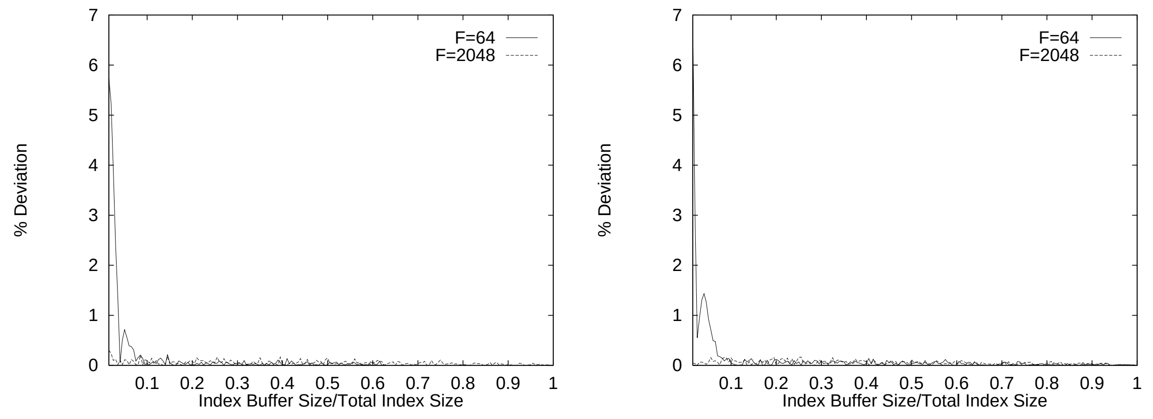


Figure 6: Relative deviation between buffer hit rate in analytical model and simulations, partitioning set 1 to the left, partitioning set 2 to the right.

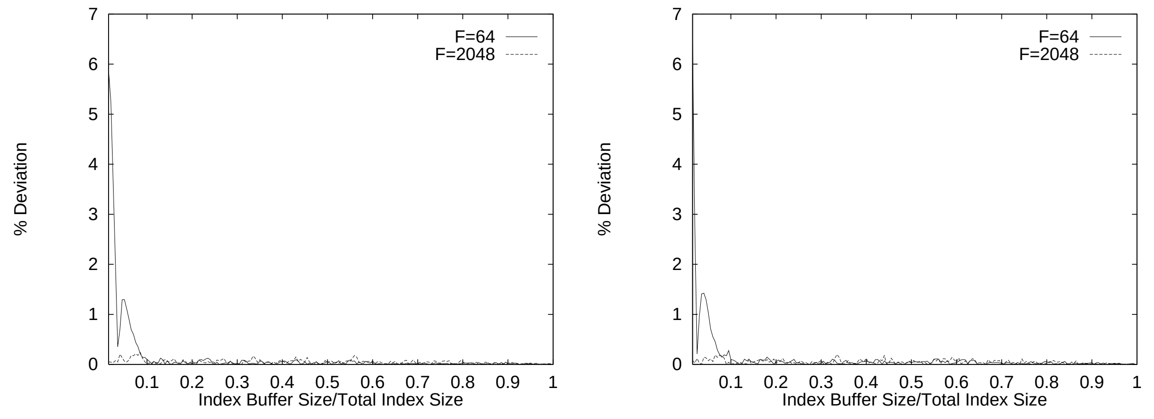


Figure 7: Relative deviation between buffer hit rate in analytical model and simulations when the no traverse strategy is used, partitioning set 1 to the left, partitioning set 2 to the right.

8 An OID Retrieval Cost Model

Our OID retrieval cost model is based on the number of disk page accesses, since this is the most significant cost factor, and common bottleneck. Disk I/O is only necessary if the requested information can not be found in memory. The model include an index page buffer, and an OE cache. It is important to note that in the general case, with an ordinary data index, an OE cache can in some situations have a very costly side effect. An increase in the OE cache size reduces the number of complete pages in the buffer, something that can make lots of costly index page installation reads necessary when the entries in the index is to be updated. In an OIDX, this issue is not so important. Index entry modification does not happen very often, usually only because of object migration or schema change. A tree based OIDX is essentially an append-only structure, the only potentially troublesome operation is deletion of objects, but index modifications because of this can be done in batch and/or as a background activity.

8.1 OID Entry Cache

A certain amount of memory, M_{ocache}, is reserved for the OE cache. If we assume the size of each entry is S_{ie}, and an overhead of S_{oh} bytes is needed for each entry, the number of entries that fits in the OE cache is approximately:

$$N_{\text{ocache}} \approx \frac{M_{\text{ocache}}}{(S_{\text{ie}} + S_{\text{oh}})}$$

Accesses to the OE 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 OE cache hit is, with p' partitions as defined in Section 6:

$$P_{\text{ocache}} = P_{\text{buf}}(N_{\text{ocache}}, N_{\text{obj}})$$

The results in the following analysis is 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 [7], but has less storage overhead,only one bit is needed for each entry. This is small enough to ignore in this analysis.

8.2 Total OID Retrieval Cost

With a probability of P_{ocache}, the OID entry requested is already in the OE 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* $P_{\text{buf\_ipage}}$ (Equation (3), with α_{Li} and β_{Li} as defined in Section 4.3). To traverse the H levels from the root to a leaf node, we need $(1 - P_{\text{buf\_ipage}})H$ disk accesses. The average cost (number of disk accesses) of an index lookup is:

$$C_{\text{indexlookup}} = (1 - P_{\text{ocache}})(1 - P_{\text{buf\_ipage}})H \tag{4}$$

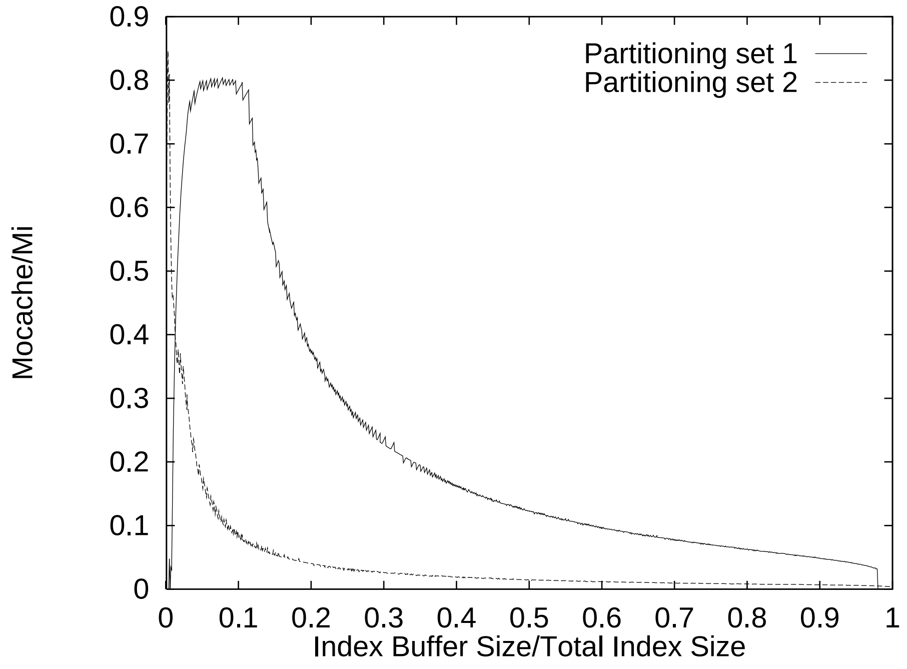


Figure 8: Optimal OE cache size vs. amount of index memory available.

9 Optimizing OE Cache and Buffer Sizes

We have now derived the equations necessary to calculate the cost of an OIDX lookup. We will in this section study in detail under which conditions an OE cache is beneficial, and how much of the index memory should be reserved as an OE cache, to get optimal performance.

The optimization problem is, given a certain amount of index memory,[4] to find the combination of M_{ipages} and M_{ocache} that gives the lowest cost, given the invariant:

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

In the rest of this report, we give OE cache size as a fraction of the total index memory size.

The study was done with databases of two different sizes: DB1 with 2 million objects, and DB2 with 20 million objects. The optimal OE cache sizes were the same for both database sizes. Unless otherwise noted, results and numbers in the next sections are from the study of DB1, with an index page size of 8 KB, and access pattern according to partitioning set 1.

9.1 Optimal OE Cache Size Versus Index Size

In Figure 8 we present the optimal M_{ocache} for different memory sizes, where optimal M_{ocache} is the one that minimizes $C_{\text{indexlookup}}$ (Equation (4)). An interesting observation, is that the optimal M_{ocache} can be relatively large, especially for medium buffer sizes.

The OE cache is most useful for capturing the accesses to the hot and medium hot areas. This is evident from the figure. As the memory size increases beyond the size large enough to capture these areas, the relative fraction of index memory useful for OE cache decreases.

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

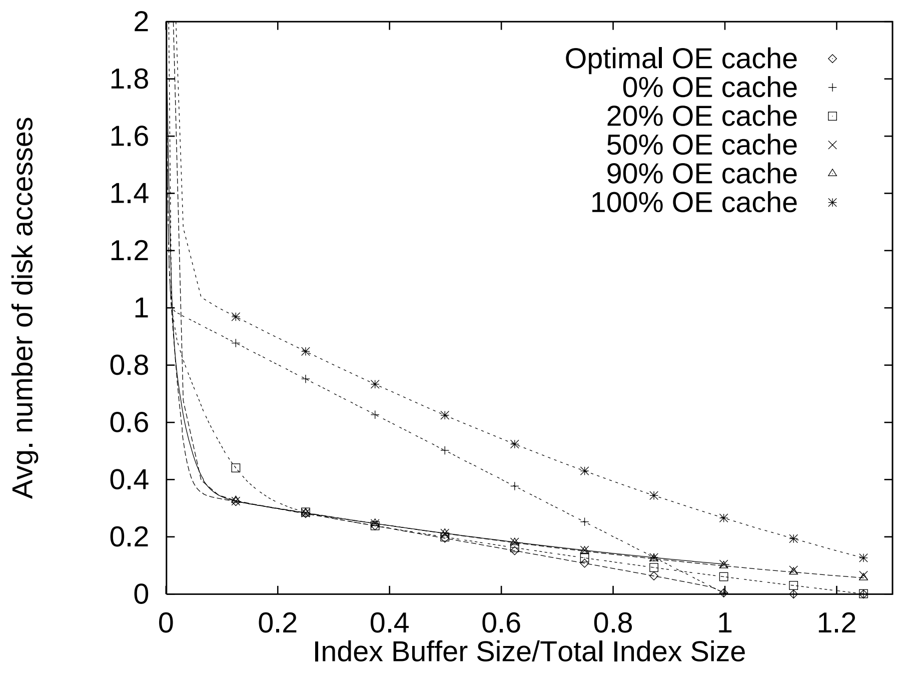


Figure 9: Cost, partitioning set 1.

When the buffer size is very small, and only a small part of the hot set fits in it, most entries in the OE cache will only be accessed once each time they are brought into the OE cache. The result is that reserving a large part of the little memory available for the OE cache can be a waste of space. In addition, to reduce retrieval cost, it will be beneficial to have as many as possible of the upper levels of the index tree in the index buffer. When the size of the index memory increases, we come to a point where most of the hot set fits in the OE cache, in addition to the most frequently accessed pages fits in the index page buffer. It is with these memory sizes (relative to the index size) we should have the largest OE cache sizes. If we increase memory even more, most of the hot set fits in the OE cache, and it is advantageous to use the extra memory for index pages. The relative size of the OE cache decreases (but the absolute size stays the same, or increases). As memory increase even more, we come to a point where the optimal OE cache size drops to zero. The reason for this, is that storing the index entries as separate entries instead of pages, incur a relatively high overhead, in the order of 50%. Stored as pages, a higher number of entries fits in the index memory, and when we come to a certain point, we get a higher buffer hit probability by using pages instead of having them in the OE cache.

9.2 Mapping Cost with Different OE Cache Sizes

Figure 9 and Figure 10 presents the average mapping costs for partitioning sets 1 and 2. It illustrates quite well the benefits of using OE caching. It also emphasizes the importance of using the optimal OE cache size. It can be seen from the figure that even though choosing a constant size of for example 50% is safe in general, if optimal performance is desired, the OE cache size should be adaptively tuned according to memory and index size. An important fact is that for most memory sizes, almost any OE cache size is better than using no OE cache at all (0% OE cache in the figures).

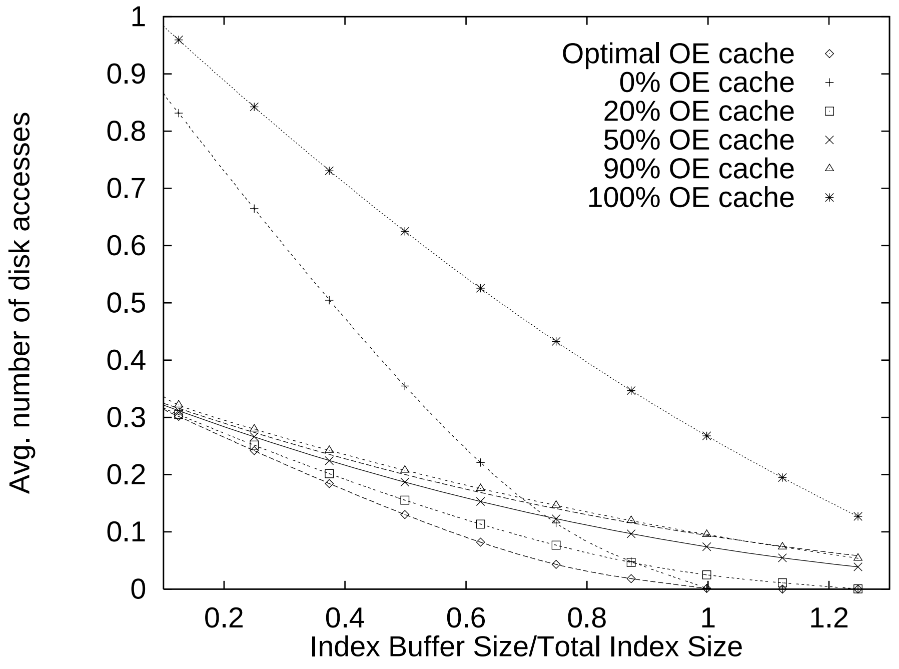


Figure 10: Cost, partitioning set 2.

9.3 The Effect of Different Access Patterns

The optimal OE cache size can be highly dependent of the access pattern. Figure 8 illustrates this for the two partitioning sets. We see that as the most narrow hot spot area (0.01%) gets more narrow (0.001%), the area where a large OE cache is needed also gets smaller.

To be able to study the effect of different access patterns, we have done some studies with only two partitions, to see how the changing sizes and access probabilities affects the optimal cache size and OID retrieval cost. Figure 11 show how the optimal OE cache size changes with different distributions. It is interesting to note that with a wide hot spot area, e.g. 70/30, it is beneficial to use almost all the available index memory for OE cache, up to a certain point. At this point, the optimal cache size drops to zero.

In the previous section, with three partitions, we had a much smaller hot spot area than in this case. The result was a drop in optimal OE cache size relative to total index memory when the OE cache was large enough to keep the hot spot, with a decline until the point where the buffer hit probability was higher with only index pages. Here we get the same result with 95/05 and 90/10 distributions, but not with the distributions with wider hot areas. The reason is that with such wide hot areas, the leaf page access distribution is almost uniform.

In Figure 12, 13, 14, and 15 the mapping cost is shown for various OE cache sizes. This show how important it is to have an optimal OE cache size, and the a fixed size OE cache typically make the OID mapping process 200 to 300 percent more costly than necessary.

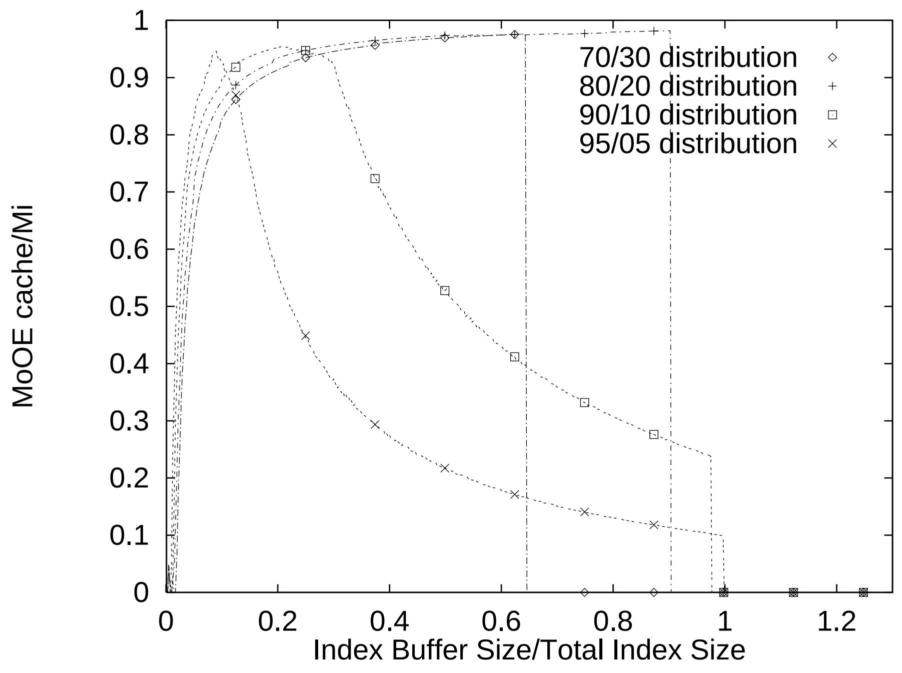


Figure 11: Optimal OE cache size vs. different amount of index memory available.

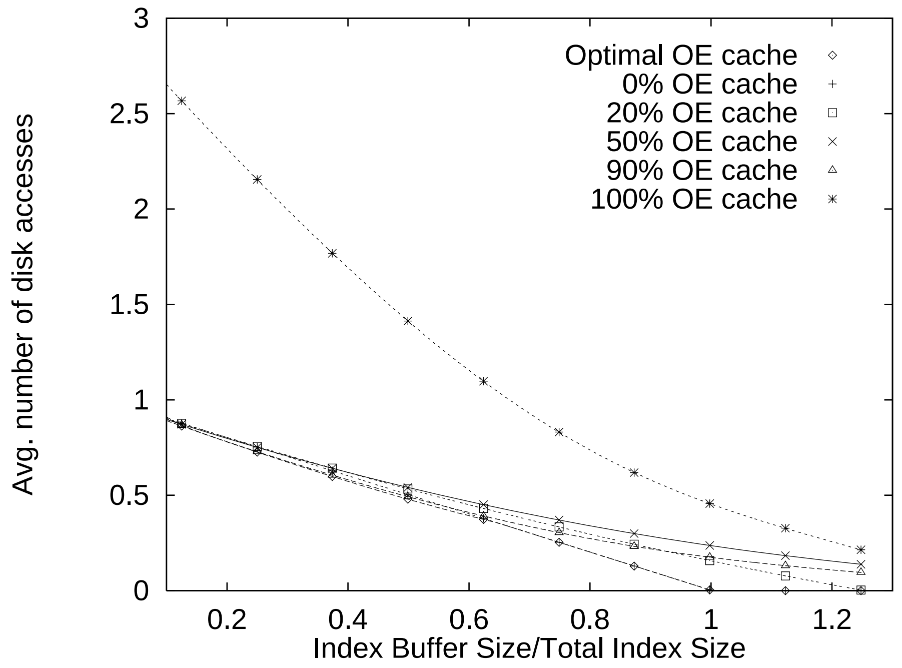


Figure 12: Cost, 70/30 distribution of accesses.

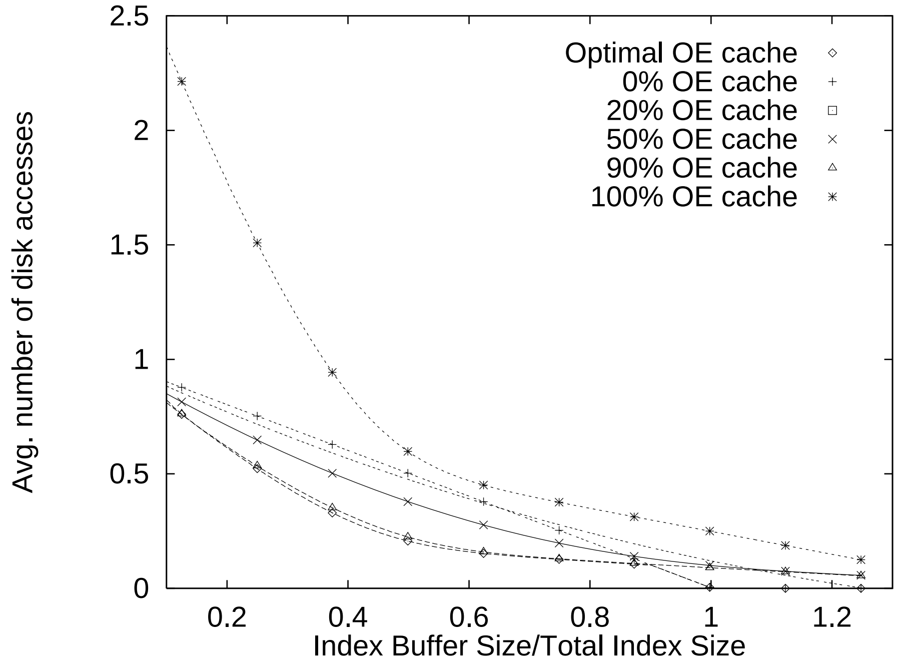


Figure 13: Cost, 80/20 distribution of accesses.

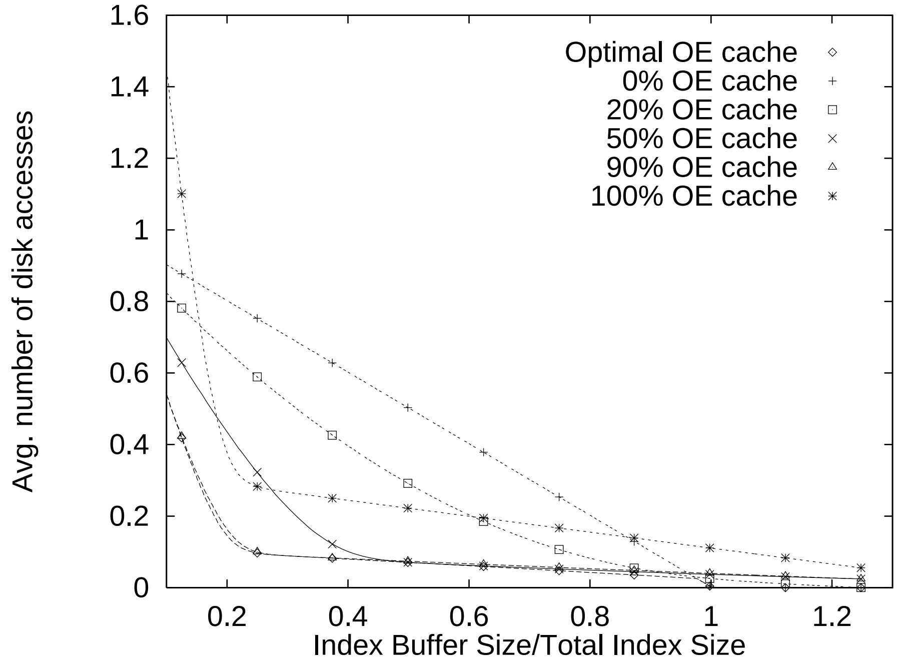


Figure 14: Cost, 90/10 distribution of accesses.

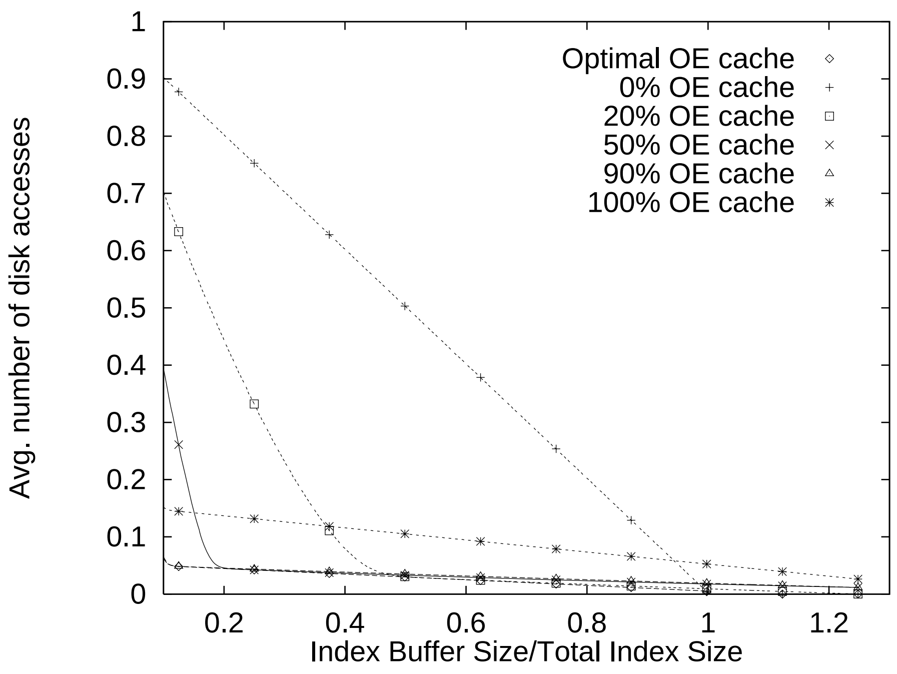


Figure 15: Cost, 95/05 distribution of accesses.

10 Conclusions

In this report, we have developed an analytical model for OIDX retrieval cost which includes index buffer and OE caching. As a part of this work, we have also develop an analytical model for hierarchical index buffer performance, based on the LRU model of Bhide et al. [2].

We use the OIDX retrieval cost model to study how memory size, index size, access patterns and different OE cache sizes affects buffer hit performance. This shows that:

1. The relative amount of memory that should be reserved for OE caching is mainly dependent of the size of the index relative to the size of the index memory, not their absolute values.

2. The relative amount of memory that should be reserved for OE caching is highly dependent of the access pattern.

3. The gain from using the optimal size of the OE cache can be very high. This is very important as the size of the available memory increases. When most of the commonly used data and index fits in the buffer, it is the infrequently accessed items, that is not resident in memory when requested, that will use most of the disk bandwidth.

The resulting models and conclusions in this report can be of valuable use for optimizers and automatic tuning tools in OODBs.

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. This, and incorporating our cost models into existing OODB query cost models, should be studied further. As for other buffer related research, buffer warm-up transients should be studied, as careful buffer managements during warm-up after system startup or workload change can improve performance considerably.

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] A. F. Cardenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5), 1975.

[4] H. T. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 14th VLDB Conference*, 1985.

[5] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of ACM SIGMETRICS*, 1990.

[6] G. Diehr and A. N. Saharia. Estimating block accesses in database organizations. *IEEE Transactions on Knowledge and Data Engineering*, 6(3), 1994.

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

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

[9] 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.

[10] W. S. Luk. On estimating block accesses in database organizations. *Communications of the ACM*, 26(11), 1983.

[11] L. F. Mackert and G. M. Lohman. Index scans using a finite LRU buffer: A validated I/O model. *ACM Transactions on Database Systems*, 14(3), 1989.

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

[13] K.-Y. Whang and G. Wiederhold. Estimating block accesses in database organizations: A closed noniterative formula. *Communications of the ACM*, 26(11), 1983.

[14] S. B. Yao. Approximating the number of accesses in database organizations. *Communications of the ACM*, 20(4), 1977.