GHOST: Fine Granularity Buffering of Index

Cheng Hian Goh[†] Beng Chin Ooi Dennis Sim Kian-Lee Tan
Dept. of Computer Science
National University of Singapore

Abstract

Buffering has all along been an important strategy for exploiting the cost/performance ratio of disk versus random-access memory. The buffering of disk pages belonging to a database has been well-studied, but literature that deals specifically with *index* buffering is scarce. This is surprising given the significance of indexes (especially B+-tree like indexes) in modern DBMSs. In this paper, we describe a dual buffering scheme for indexes, called GHOST, in which part of the buffer is used to maintain popularly used "paths" of the B+-tree index, while the remainder is devoted to maintaining a Splay-tree with pointers to leaf pages containing frequently used leaf pages. This scheme allows us to maintain pointers to leaf nodes long after the paths leading to the leaf nodes have been replaced, thus maintaining "ghost" paths to the nodes. In addition to describing the search and maintenance operations for the GHOST buffering scheme, we also conduct a series of experiments in which it is shown that GHOST outperforms the best existing schemes (ILRU and OLRU) by impressive margins for almost all pragmatic query workloads.

1 Introduction

Despite the fact that modern computer systems are equipped with an abundance of main memory, the latter remains a scarce resource with increasingly sophisticated (and large) software and the rapid buildup of huge data sets. In particular, although database servers are now routinely equipped with between 512 MB and 2 GB of main memory, it is not uncommon to find databases consisting of terabytes of data. Moreover, any available memory will have to be fragmented among large numbers of concurrent transactions. For example, at the National University of Singapore, the university database is managed by Oracle 8.05 running on a powerful 4-way V2200 HP machine with 2 GB of memory. Applications range from student result management system, human resource to fund management systems. It has a user base of 26,000 and a database size of about 100 GB. Many of the relations are indexed on attributes that are frequently queried. Despite the powerful configuration and proper database tuning, due to the high number of applications and stored procedures, we have observed that at peak, the memory resources and processors are fully utilized. In short, a good buffer management system that is effective in exploiting differential access speed of main memory (versus disk) remains a key determinant of system performance and throughput.

In the past, a variety of different buffering schemes for pages of a database have been proposed. Early works were adaptations from research in operating systems [7], such as the LRU and its variants [10], the Hot Set model [13], and other mutations (e.g. DBMIN [5]). Interestingly, very few work have been reported on the buffering of database indexes themselves. To the best of our knowledge, only three such schemes have been reported in the literature: the ILRU and OLRU [12], and the extensible buffer mechanism described in [4].

In this paper, we studied the management of buffers for large B+-tree-like indexes. Besides being the most widely used indexing scheme, the B+-tree index [6] has also been shown in recent works to provide superior performance in managing high dimensional data [3, 11]. Our goal is to further improve the performance by minimizing the number of page faults for fetching index pages from large B+-trees. The novelty of our proposal lies in the use of a dual-buffering scheme; specifically, one part of the buffer space is used as an

---

[†]Deceased on 1 April 1999.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

ILRU buffer that allows popular paths of the B+-tree to remain in main memory, while the rest is devoted to caching pointers to leaf pages, and are organized into a Splay-tree [14]. This allows previously popular paths of the B+-tree to be evicted from the ILRU buffer, while keeping the pointers to leaf nodes directly, making it appear that "ghost" paths now exist in the main memory buffer.[1]

We also studied the sensitivity of the buffering approach to the proportion of memory devoted to the ILRU buffer (and as a consequence, those available to the Splay tree) and the efficacy of GHOST under different levels of data skew. It is shown that GHOST outperforms existing index buffering schemes by impressive margins under most circumstances, regardless of the amount of memory available and different amount of data skew.

The remainder of this paper is organized as follows. In section 2, we introduce the case for buffer management of indexes and describe briefly the various techniques proposed in the literature. This is followed by a description of the splay tree structure in section 3, and the GHOST strategy for buffer management in section 4. In section 5, we present results of the experimental studies using the proposed buffering scheme under different retrieval patterns, and compare these results to existing buffering strategies. Finally, we summarize our contributions in the last section and describe some work in the pipeline.

2 Buffer Management for Indexes

The management of buffers *for indexes* has received considerably less attention compared to the problem of buffer management in general. Throughout this paper, we assume that a B+-tree index [2] is used. Nonetheless, the analysis and experimental results in this paper can be easily extended to most other hierarchical index structures.

To the best of our knowledge, only two distinct set of works have been reported in the literature [4, 12]; specifically, the replacement policies *ILRU* (Inverse LRU) and *OLRU* (Optimal LRU) proposed in [12], and a more generic *priority-based* approach proposed in [4]. These approaches are described briefly in the next three subsections.

2.1 Inverse LRU (ILRU)

The ILRU replacement policy is formulated with the following observation in mind. Access to index pages involves the traversal of a tree from the root to the leaf pages. Suppose p is the parent page of the set of of children denoted by c. To access a child c_i ($\in c$), p must be first accessed. Hence, the sum of accesses to children in c cannot exceed the total number of accesses to p. Consequently, it is always suboptimal to replace p, if any of the $c_i \in c$ are still in the buffer.

Let B be the number of buffers allocated for supporting the index, and suppose L is the height of the B+-tree (i.e., the number of traversals needed to reach a leaf node from the root, plus one). It should be clear that whenever $B < L$, the classic LRU policy will perform poorly, since the pages which are nearest to the root will be swapped out first.

The Inverse LRU (ILRU) policy modifies the LRU policy by simply reversing the order in which pages are scheduled for replacement. Hence, when a page p at level i (the root being level 0) is accessed, it is not placed at the top of the LRU stack but at its i-th position: thus, the root will be placed at position 0 (the top of the stack). Index pages being accessed are always appended. In the exceptional situation when the stack has $B < L$ buffers, the currently referenced page at level i ($> B$) is placed at the B-th position. This means that whenever no more buffers are available, the page at position B of the LRU-stack will be replaced. This strategy guarantees that top level pages of a B+-tree always have higher priority compared to those further from the root. As an example, a buffer with $B \geq 2$ is sufficient to keep the root in the buffer at all times.

2.2 Optimal LRU (OLRU)

In the case of the Optimal LRU (OLRU) strategy, the index pages are logically partitioned into L independent regions, each managed by a local LRU stack. The number of buffers allocated to region i is given by T_i and is estimated by Yao's function [15]. There are two distinct cases that need to be considered. If $B \geq T(= \sum_{i=1}^{L} T_i)$ then all of the regions can be given a full allocation. If however $B < T$, then there exists $j \leq L$ such that $\sum_{i=1}^{j-1} T_i < B < \sum_{i=1}^{j} T_i$. In which case, we allocate T_i buffers to regions T_i ($i = 1, 2, \ldots, j-1$) and the remainder $B' = (B - \sum_{i=1}^{j-1} T_i) - 1$ to region j. In other words, three types of regions may exist in the buffer for a given buffer size: *non-deficient* regions where each is allocated the full set of T_i buffers; *coalesced* regions which share a single unallocated buffer; and finally a single *deficient* region that is allocated B' buffers.

Whenever an index page at level i is accessed, it is kept in region i of the buffer (or, in the case of coalesced regions, in the single free buffer). Buffers in a given region are managed using the LRU policy (though a random replacement policy may work equally well). Under the assumption that leaf pages are accessed with the same probability, this allocation is optimal because available buffers are allocated to the index pages according to their reference frequency. Sacco [12] has also shown that similar results can be obtained even if the access probability is skewed (such as when a Zipf distribution is used).

[1] The acronym for the proposed buffering strategy is also a pun on the authors' names.

2.3 Priority-Based Index Buffering

In the priority-based buffering scheme proposed in [4], each page in the buffer is assigned a priority value which may be dynamically modified to reflect its "replacement potential" relative to other buffered pages brought into memory. Pages present in the buffer is denoted as either *useful* or *useless*: a buffer page is considered useful if it is to be re-referenced again (in the context of the same query) and useless otherwise. This information is known because of the predictability of the reference pattern. Within each set, the pages can be further assigned priority values based on the level number of the page with reference to the *anchor page* (the index page furthest from the root that contains the entire range of leaf pages required by the query). More specifically, priority values are assigned as follows:

- useful pages are assigned higher priority than useless pages;
- among useless pages, higher priority is assigned to pages nearer to the root (since these are more likely to be accessed in *subsequent* queries);
- among useful pages, higher priority is given to more recently used pages since they will be re-referenced sooner because of the depth-first traversal reference pattern.

Notice that this priority scheme is dynamic since the priority of a page may decrease as the B+-tree is traversed; for example, a useful page (descending from the anchor page) becomes useless after the entire subtree rooted at that page has been traversed.

Under the prescribed priority scheme, the buffer page with the least priority is selected as the victim whenever the buffer runs out of room. This replacement strategy can be understood as a hybrid scheme representing a combination of the LRU and MRU replacement policies. Specifically, useful pages are managed in an LRU manner (favoring pages nearer to the leaf), and useful pages are managed using a MRU policy (favoring pages nearer to the root).

2.4 Discussion

The performance of ILRU and OLRU are compared in a preliminary study reported in [12]. Both schemes are shown to be better than the classic LRU.

We observed that in the case of B+-trees, the priority scheme in [4] does not present a distinct contribution for the following reasons. For key-probes, it degenerates into ILRU since all pages are useless. In the case of range queries, there is no occasion for the index structure to be backtracked (to the anchor node) since adjacent leaf pages are linked. (Note that this is only true for this context; in a more generic tree-based indexed, such as the R-tree [9], the priority scheme will actually behave differently from ILRU.)

3 The Splay-Tree

The Splay tree is a self-adjusting binary search tree introduced in the mid 1980s [14]. It is shown in [14] that although the operations (e.g., search, insert, delete) performed on a Splay tree is not necessarily individually efficient, they are guaranteed to be so over a sequence of operations. In other words, it has been shown that the *amortized cost* of a sequence of operations of a Splay tree is bounded by $O(log\ n)$. This behavior is achieved by a *splaying* operation which allows the Splay tree to be restructured after each operation, so that subsequent operations can be accomplished more cheaply.

The splaying operation consists of a sequence of rotations in such a way that the node containing the key (or a node which would be a neighbor of this key if the latter is not in the tree) ends up being the root of the splay tree. In performing the Splay operation, three cases can be distinguished depending on the node R being accessed and its parent Q and grandfather P (if any).

case 1: Node R's parent is the root.

case 2: *Homogeneous configuration.* Node R is the left child of Q which in turn is the left child of P; or both are right children.

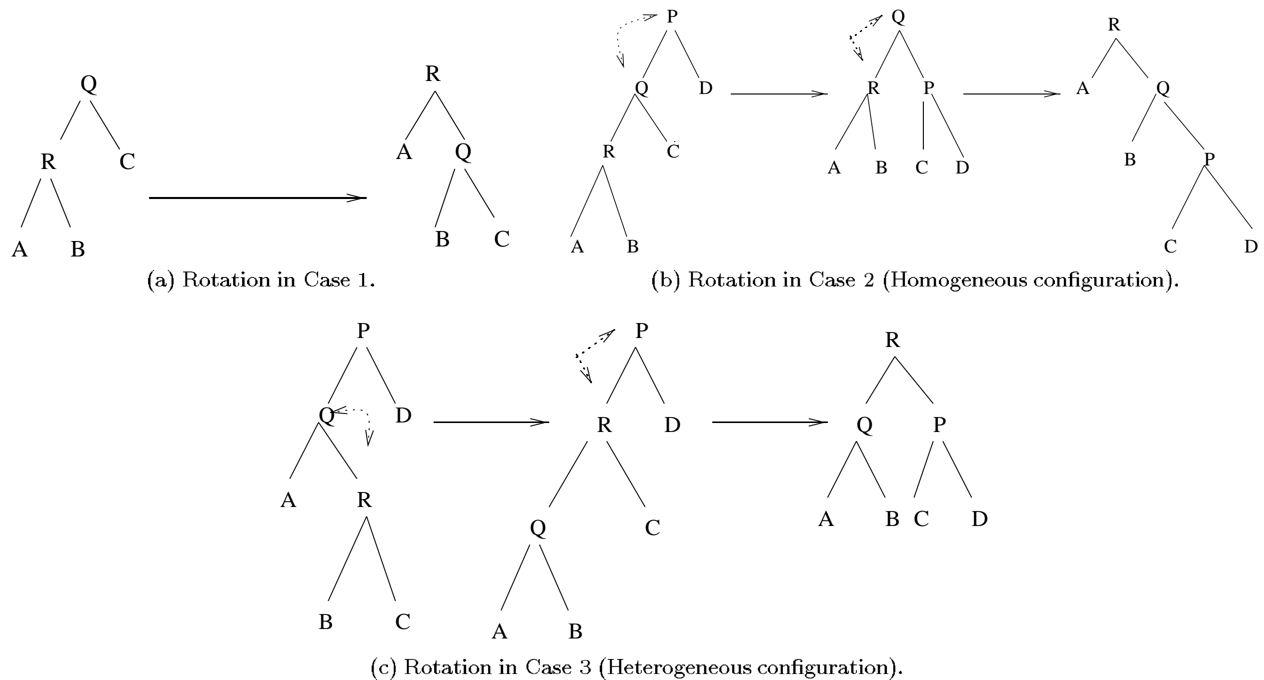
case 3: *Heterogeneous configuration.* Node R is the right child of parent Q which in turn is the left child of P, or vice versa.

Each configuration presents different opportunities for elevating node R to the root. Figure 1 shows the animated rotation associated with each configuration. The abstracted algorithm is given in Figure 2. One readily observed property of the Splay tree is that *the most recently observed key (being searched on or inserted) will float to the top of the tree, while the remainder percolates down to lower parts of the tree.* Because of this property, the search cost for the same node is greatly reduced.

4 The GHOST Index Buffering Scheme

In the classical management of buffers for indexes, it is common to adopt index pages (which includes both internal and leaf pages) as a unit for buffering: i.e., an index page is either in the buffer, or it is not. In this section, we describe a dual-buffering strategy which consists of two different data structures

- a conventional index buffering scheme (in our case, the ILRU scheme) that allows recently used paths in the index to be cached; and
- an appropriate in-memory data structure (in our case, the Splay tree) that allows pointers to leaf



(a) Rotation in Case 1.

(b) Rotation in Case 2 (Homogeneous configuration).

(c) Rotation in Case 3 (Heterogeneous configuration).

Figure 1: Animation of rotations for different Splay cases.

nodes to be stored (accompanied with their respective key ranges).

Notice that in the second case, the leaf node pointers may remain in the main memory even long after the paths have been evicted. This creates the impression that a "ghost" path to the leaf node exists and hence the name of the index buffering scheme.

The proposed dual buffering scheme is motivated by the following observations. As the memory resident data structure stores pointers to leaf nodes, given a fixed amount of memory, it can store more pointers than that under a conventional scheme that stores paths. By retaining as many pointers to frequently accessed leaf nodes as possible, we hope to reduce the access cost to these nodes to only a single page fault. For the conventional scheme, because of the need to store paths, the number of index nodes that can be retained in the main memory is severely limited. On the other hand, purely storing pointers to leaf nodes is not expected to perform well also. This is because in the event of a cache miss, the entirety of the path (from root to the leaf node) has to be accessed. The conventional scheme's ability to retain the path can avoid this. Thus, a dual buffering scheme is expected to be superior. Our experimental study in section 5 confirms these observations.

4.1 Key Data-structures in GHOST

There are three key data structures in GHOST:

- a B+-tree index;
- a ILRU buffer; and
- a mutated Splay-tree.

The B+-tree is a hierarchical index constructed over the data. It is normally too large to be stored in the main memory and relies on some buffering scheme to keep the commonly traversed index records in the cache. The ILRU scheme is chosen because it gives greater priority to the pages of the B+-tree nearer to the root. Furthermore, it is simpler and is reported to yield better performance than schemes that are known.

The ILRU buffer operates exactly as described in [12]. By caching popular (frequently and least recently used) paths in the tree, the ILRU scheme provides a method for caching the paths leading to the data nodes.

Finally, we adopted the Splay tree as our in-memory data structure. The Splay tree used by GHOST differs from the one described earlier in two aspects. First, despite the (relatively) large amount of main memory available, the latter resource is finite. This requires an eviction policy: in our implementation, a leaf node is chosen randomly to make room whenever necessary. Recall that in a Splay tree, data at the leaf nodes are the least recently used, and hence randomly picking a leaf node is a reasonable heuristics. In fact, it is this property that the Splay tree is preferred over other *height-balanced structures* (such as AVL trees [1] and Red-Black trees [8]). There is no mechanism in

Algorithm Splay:

```
while R is not the root
    if R's parent is the root
        rotate R about Q
    else if R is in a homogeneous configuration with its predecessors
        perform a homogeneous splay, first rotate Q about P
        and then R about Q
    else /* R is in heterogeneous configuration with its predecessors */
        perform a heterogeneous splay, first rotate R about Q
        and then about P
```

Figure 2: Splay algorithm.

these structures to facilitate an efficient eviction policy without maintaining additional information. This, however, will consume the already scarce storage resource and hence is expected to be less effective than the Splay tree. Second, the node structure of our Splay tree differs slightly: as shown in Figure 3, ours is laden with a key range as opposed to a single key value, and an additional pointer to a leaf page.

Figure 3 provides a pictorial representation of how the different data structures are related to one another. The figure contains two parts: the top portion shows that the main memory is split into two regions, one for the splay tree and the other for the ILRU; the second portion shows the disk-based B+-tree structure. The B+-tree shown stores the data records at its leaf nodes. As shown, the splay tree buffers store the frequently accessed leaf pages, and the ILRU buffers manage frequently traversed paths (i.e., the internal nodes of the B+-tree). We note that the leaf nodes are not kept at the ILRU buffer. This is because it will be kept in the Splay tree where the search process begins (see the Search algorithm in the next section).

4.2 GHOST operations

As in any data structures, the three primitive operations that have to be addressed are: search, insert, and delete. In our case, we are interested in the effects of these operations and their implications for the GHOST scheme.

Search

Figure 4 describes the algorithm for searching via its search key. There are two cases to consider. The first is when the pointer to the leaf page is in the Splay tree: in which case, only one I/O for fetching the leaf page from disk is needed. Searching the Splay tree will bring the relevant Splay tree node to be the root. This ensures that frequently and permanently used leaf pages have corresponding entries near the top of the Splay tree.

Second, if the required key cannot be found in the Splay tree, the B+-tree will be searched directly. The presence of an ILRU buffer allows popular index pages to be obtained without additional disk I/O.

Algorithm Search(Key)

```
Search Splay tree for a node where the key
        is covered by the key interval
if found, return pointer to leaf page
else (/* not found */)
        Search B+-tree, with the aid of ILRU buffer
            to reduce the amount of I/O
        When the leaf page is found
            create a new node for it
            insert it into the Splay tree
            Return pointer to leaf page.
```

Figure 4: Search algorithm for the GHOST scheme.

Insertion

Insertion of a new data item in a DBMS is preceded by a Search operation. This means that by the time the appropriate node is located, there will be a Splay node entry for the corresponding leaf node in the root of the Splay tree. There are now only two situations to worry about:

- When the boundary values of the leaf node have been changed. For instance, in inserting a data item with key value 96 into a leaf node having key interval [100,145], we are required to update the key range in the Splay tree root to [96,145].

- When the insertion causes a node split in the leaf node. For example, if insertion causes a node split with node A (interval [96,120]) and B [121,145]), the corresponding key interval in the Splay tree will have to be changed to reflect one interval, and a new node is inserted to reflect the other interval.

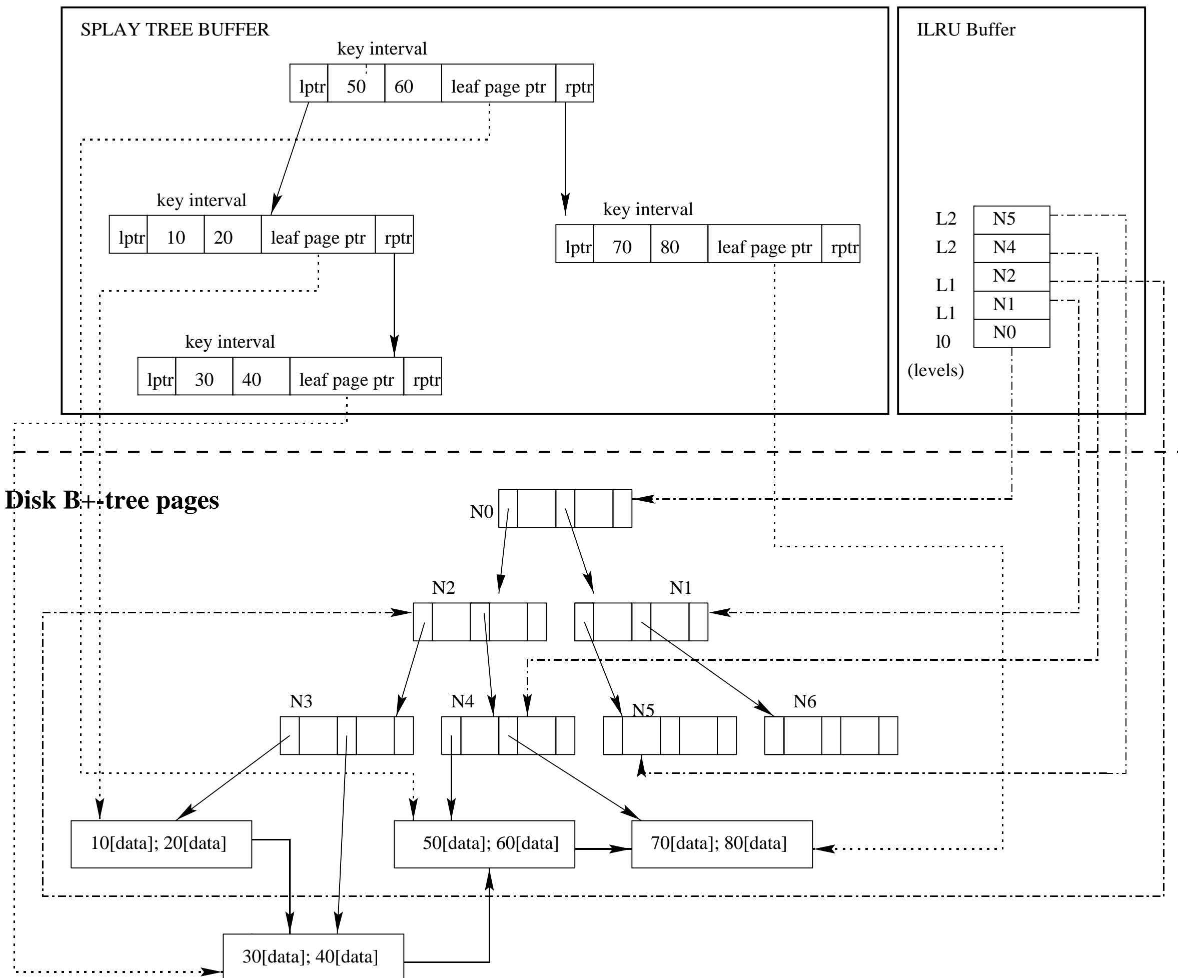


Figure 3: A graphical representation of the GHOST scheme.

Deletion

Like insertion, deletion requires a prior search operation. The situations requiring attention are symmetrical to the ones in Insertion:

- When the deletion causes a change in the key intervals as a result of deleting data items that are boundary key values in the Splay tree nodes. This requires changing the Splay tree nodes to reflect the new ranges.
- When deletion causes two leaf pages to be merged together, thus requiring a change to the key interval of one Splay tree node, while the other can be deleted.

5 Experimental Study

We present in this section an experimental study of the GHOST strategy proposed in this paper. There are two performance metrics. The first is the hit ratio which is defined as the ratio obtained by dividing the number of requests into the number of page faults (i.e., disk I/O operations). The second is the number of disk I/Os missed in a sequence of 500,000 disk queries made.

The data set we used in our experiments consists of 10 million unique integers drawn randomly from the interval $[1..100M]$. These numbers are inserted into a B+-tree at random prior to the conduct of any experiment. Table 1 summarizes the system parameters that are used. We assume that each data record contains 200 bytes and is stored in the leaf of the B+-tree. The resultant B+-tree has four levels with $1, 61, 6868$ and $769, 230$ nodes at levels $1, 2, 3$ and 4 respectively.

We experimented with three different types of queries: uniform, skewed-clustered, and skewed-unclustered. The difference among the last two is that while queries are skewed towards certain datapoints,

Parameter	Default Values	Variations
System Parameters		
page size	4K	
buffer size	128 pages	16-1024 pages (or 64K-4M)
percentage of buffers allocated to Splay tree	50%	0-100%
disk pointer size	16 bytes	
data key size	8 bytes	
in-memory pointer	4 bytes	
Splay tree parameters		
size of node	40 bytes	
Database Parameters		
number of records	10 million	
size of record	200 bytes	
Query Parameters		
query type	point	range
query distribution	skewed-unclustered	uniform, skewed-clustered
no. of queries	500,000	

Table 1: Parameters and their values.

the latter's spread of skewness is across a greater range, i.e., the number of "hot" pages is larger for the latter. This behavior is demonstrated in Figure 5. Notice that both data query points in skewed-clustered and skewed-unclustered distributions are identical (Zipf at 0.01), but the latter is randomly scattered.

The first three sets of experiments examine the performance of our proposed buffering scheme, GHOST, against ILRU and OLRU, under different workloads as total available size for index buffering are varied. The priority scheme of [4] will not be examined for reasons highlighted in section 2.4. In the GHOST scheme, 50% of the available memory is devoted to the Splay tree and the remaining is reserved for buffering index nodes under the ILRU strategy.

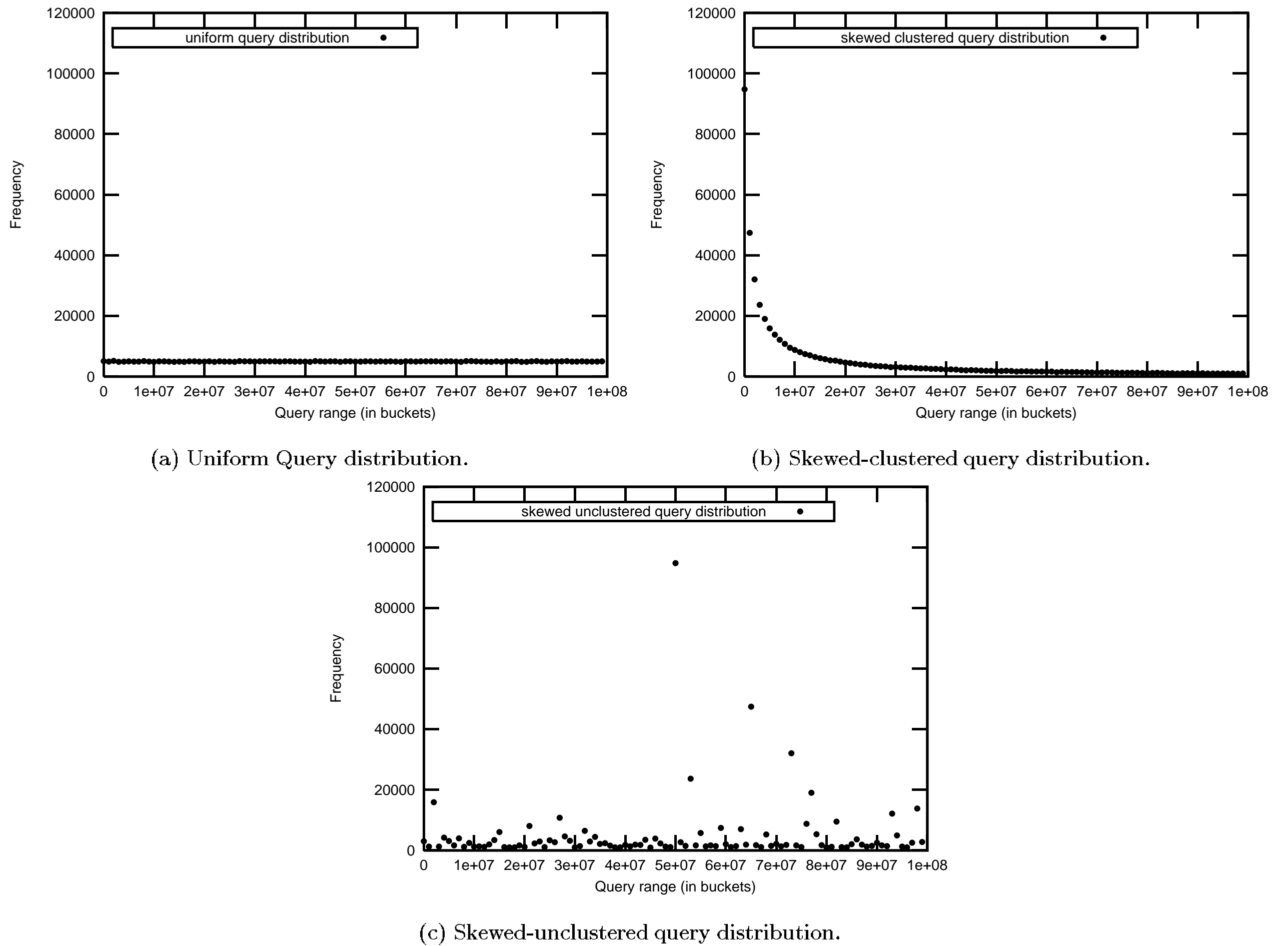
5.1 Effect of Workloads

Experiment 1

In the first experiment, we examine the performance of different buffering schemes under *uniform workload.* Intuitively, uniform workload is the worst case scenario for the GHOST scheme. This is because, under uniform workload, all records are equally likely to be accessed, and so the benefit of retaining pointers to "frequently" accessed leaf pages (all pages are equally frequently accessed!) significantly reduces. Figures 6(a) and (b) show the results under which total memory size is varied from 16-1024 pages. As expected, the hit-rate under all schemes increases with the buffer size (see Figure 6(a)). Furthermore, we observe that the hit-rate under OLRU and ILRU are roughly the same. While the GHOST scheme performs close to OLRU and ILRU for small buffer sizes ($<$ 128 pages), it is twice as well for large buffer sizes. Figure 6(b) shows the number of cumulative number of page I/Os under each buffering scheme. As is predicted, all the numbers decrease with increasingly larger buffers. However, we observe that when the buffer size is small, the GHOST scheme actually incurs more I/Os (despite its higher hit rate). Upon investigation, we found that this is attributed to the high page fault when there is a cache miss in the Splay tree. For small buffer sizes, the space allocated to the ILRU is too small to be useful. As a result, any cache miss will result in the entirety of the search path being accessed. As the buffer sizes increase, the GHOST scheme's ability to retain more pointers (than the ILRU and OLRU) still pays off and it takes the lead once again.

Experiment 2

Figures 7(a) and (b) show the performance of the three buffering schemes under a skewed-clustered query distribution. Unlike Experiment 1 where the query is uniform (a.k.a. random), we expect queries to be skewed along a given key cluster. This suggests that better performances can be expected. Indeed, the GHOST scheme performs better than the closest contender in both hit rate and total I/O count. For I/O count, the gain is about 25%. This is considered significant since under a clustered skewed distribution all the frequently accessed leaf nodes will be collected in close proximity, which a simple ILRU-type scheme can be expected to perform well as the same paths along the index tree is traversed frequently to reach that data cluster.



(a) Uniform Query distribution.

(b) Skewed-clustered query distribution.

(c) Skewed-unclustered query distribution.

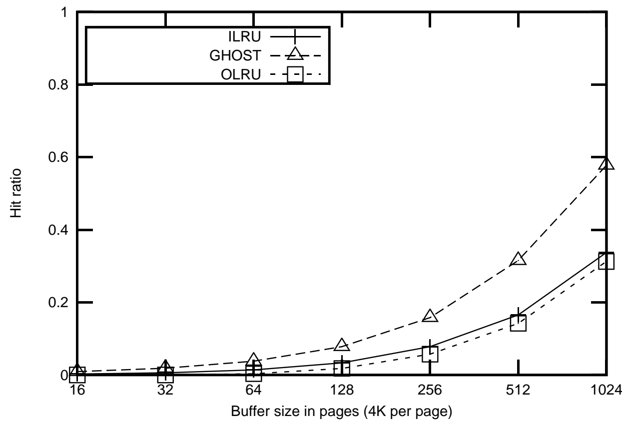
Figure 5: Different Query workloads and their visual representation.

Experiment 3

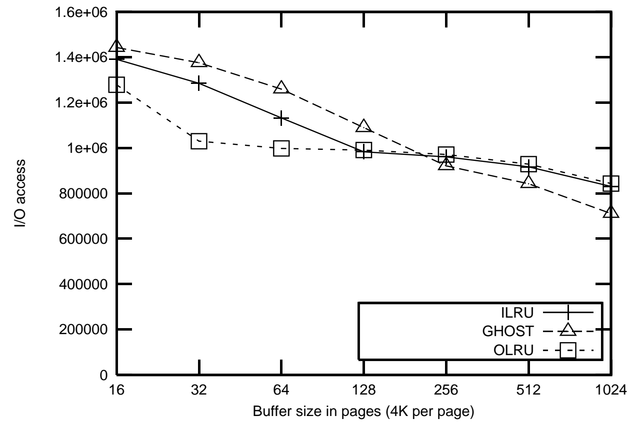
In this experiment, we compare the three schemes under skewed-unclustered distribution. Unlike the skewed-clustered distribution, there are now multiple non-contiguous active clusters. As a result, there will be many more such paths and it will become unlikely that all or these paths can be cached. Thus, we can expect the performance of ILRU and OLRU to be poorer. This is confirmed in our result shown in Figure 8. On the other hand, in the GHOST scheme, the problem is corrected through the use of the Splay tree nodes, whose nodes store only pointers to leaf nodes. GHOST outperforms ILRU and OLRU by up to 30% in terms of I/O. The results clearly demonstrate the impressive performance of the GHOST scheme over the rest.

5.2 Effect of Range Queries

We also conducted an experiment to evaluate the relative performance of the three buffering schemes on range queries under a skewed-unclustered workload (skew factor of 0.01). Here, we fixed the total number of memory pages to 256 (i.e., 1 MB of memory), and 50% of the space is allocated to the Splay tree. For a range query with interval $[r1, r2]$, it is evaluated by searching for the leaf node that contains $r1$, and the chain of leaf nodes are followed to retrieve all records whose key values are in the interval. In this experiment, the queries are generated such that it retrieves an average of four leaf nodes. We vary the percentage of range queries from 0% (all point queries) to 100% (all range queries). The result is shown in Figure 9. As shown, the hit ratio is not affected by the percentage of range queries. However, all schemes' I/O counts increase with the percentage of range queries. This is expected since more data are being accessed. The interesting point to note is that the GHOST scheme remains the best scheme, and outperforms the other scheme by up to 20% of the total I/O count. We note that this is lower than the result for point queries. This can be explained as follows. The total I/O count comprises two components: internal nodes and leaf nodes. For all the schemes, the cost to access the leaf nodes are the same and is larger for range queries than point

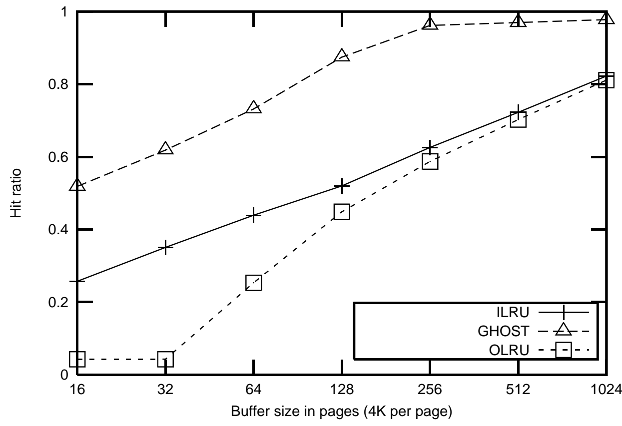


(a) Hit rate.

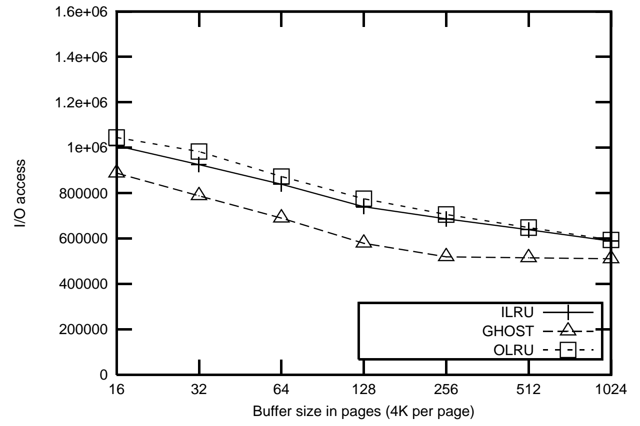


(b) I/O cost.

Figure 6: Query hit rate and page I/O of different buffering schemes under uniform query workloads.

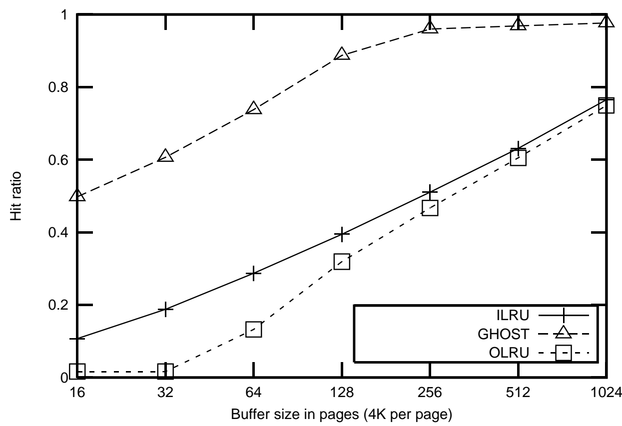


(a) Hit rate.

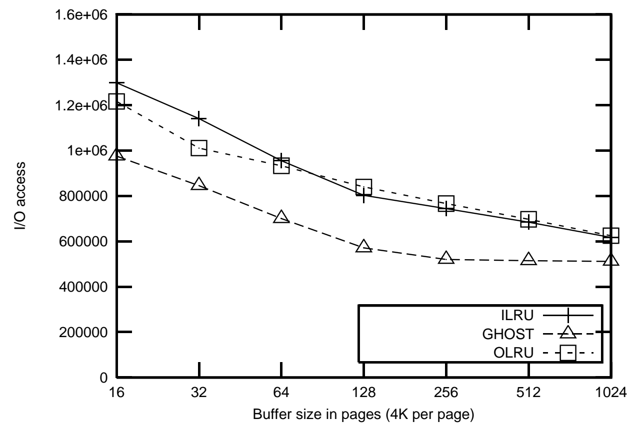


(b) I/O cost.

Figure 7: Query hit rate and page I/O of different buffering schemes under skewed clustered distribution.

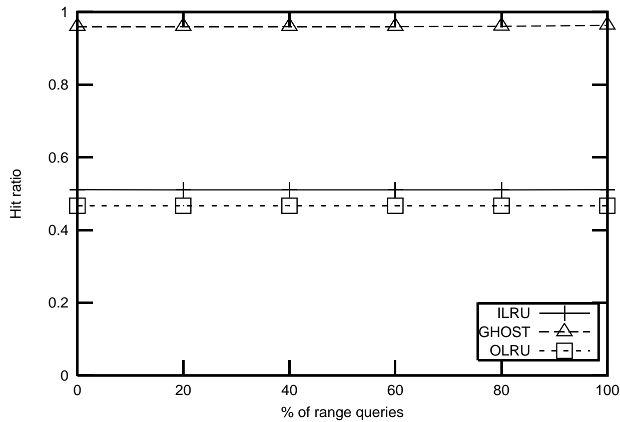


(a) Hit rate.

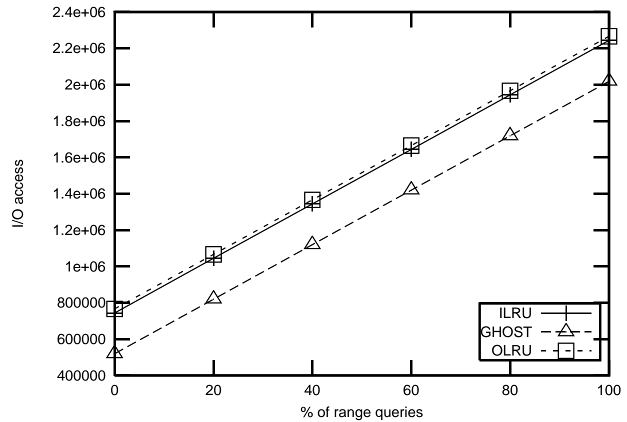


(b) I/O cost.

Figure 8: Query hit rate and page I/O of different buffering schemes under skewed unclustered distribution.

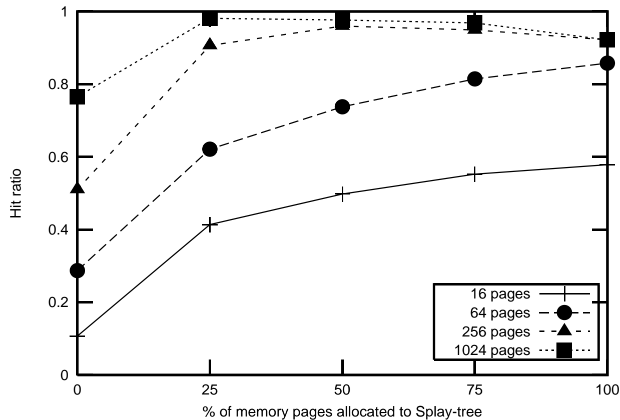


(a) Hit rate under different % of range queries.

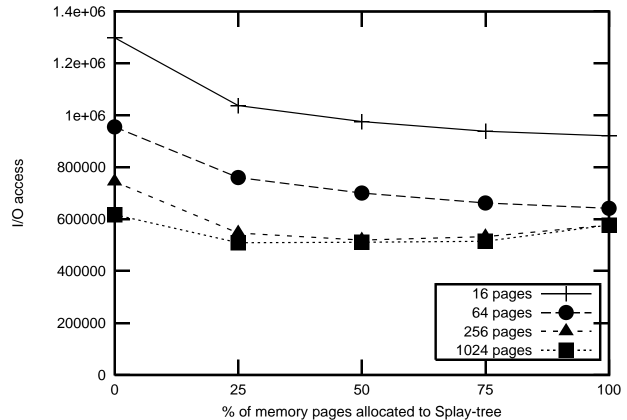


(b) I/O count under different % of range queries.

Figure 9: Results for range queries.



(a) Hit rate under different memory allocation.



(b) I/O count under different memory allocation.

Figure 10: Results under skew unclustered for varying memory allocation and distributions.

queries. The buffering schemes, however, only benefit internal nodes. Thus, we can expect the gap between GHOST, and ILRU and OLRU, to be narrower as the range increases. However, it is uncommon to have large number of range queries with large ranges. Thus, GHOST is a promising alternative buffering scheme to ILRU and OLRU.

Before we leave this section, it is worth noting that the experimental study is based on a naive evaluation strategy for range queries, i.e., it can be considered a worst case scenario for GHOST. In fact, a cleverer scheme is to search for a node in the Splay tree that intersects the query interval (instead of one that contains the left boundary value). In this way, the chances of a hit in the Splay tree is higher. This will, however, requires us to modify the leaf nodes to be chained backward as well as forward. Thus, by hitting a node that is in the middle of a range, we can use the backward and forward pointers to traverse the leaf nodes. We are currently investigating this scheme.

5.3 Effect of Memory Allocation

In the preceding experiments, we have split the total buffer size evenly between the ILRU buffer and the Splay tree. One of our concerns is whether and how GHOST's performance is affected by the allocation of memory between the Splay tree and the ILRU buffer under varying amount of available total memory.

While we do not expect the GHOST scheme to be superior when 100% of the available memory are allocated to the Splay tree, we do conjecture that the performance of GHOST is likely to be better if a larger portion of buffers are allocated. We verify this with an experiment, where we plot the performance metric against different internal allocation under different allocations of total buffer memory. Figure 10 shows our

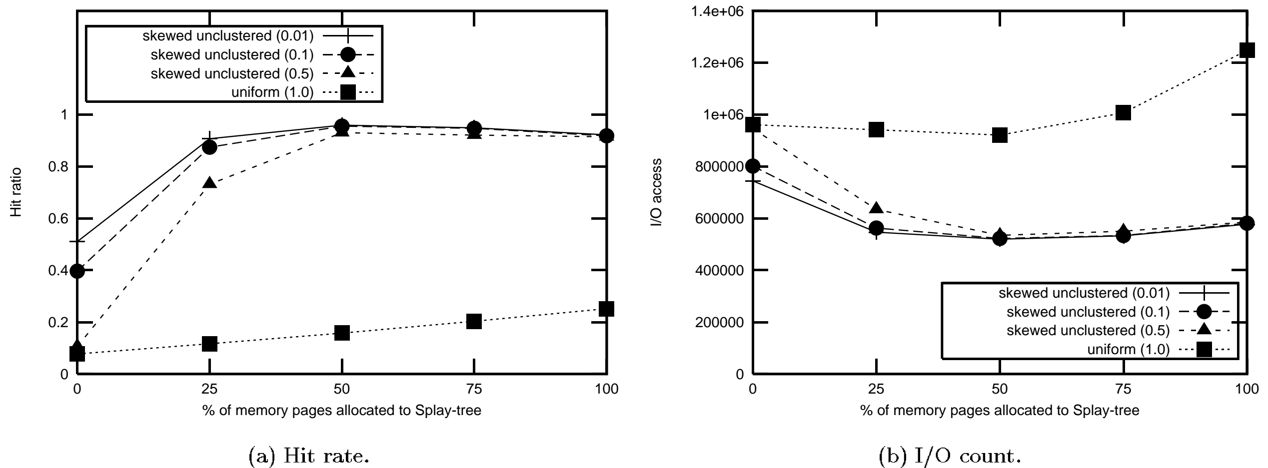


Figure 11: Results under skew unclustered for varying Zipfian factor.

empirical findings for four different buffer sizes: 16, 64, 256 and 1024 pages. The evidence reveals more then just a simple relation; instead, it suggests that when the total available memory is small (e.g. in the 16-page case), allocating more memory to the Splay tree will give a sharp boost to the performance. This further confirms our result and observation in earlier experiments. On the other hand, with a large buffer size, the performance hits the performance limit rapidly even with small allocations to the Splay tree. We also observe that assigning too much buffer may not be helpful. For very large buffer size, the Splay tree is caching not only frequently accessed pointers, but also those that are not frequently accessed. Pointers in the latter category are often replaced before they are being accessed again, thus caching them only results in fewer space being allocated to the ILRU buffers (which in turn restricted the number of paths being retained in the ILRU buffers). As a result, the overall performance degrades a little.

We also studied the effect of the distribution of queries under the skewed-unclustered workload, i.e., how the value of the zipf factor affects the performance of GHOST. In this experiment, we fixed the total memory at 256 pages. Figure 11 shows the results of this experiment. When the Zipf factor is 1.0, all leaf pages are equally likely to be accessed, i.e., the skewed-unclustered workload degenerates to the uniform workload. In this case, it becomes more beneficial to allocate more memory to the Splay tree. On the other hand, a small Zipf factor means more queries are accessing a certain set of "hot" pages. Our results show that the hit rate and I/O performance improves as the Zipf factor reduces. This is because the chances of finding the pointers to the hot pages in the Splay tree increases with more queries accessing them. However, unlike the uniform workload, we note that we do not need to allocate too much memory to the Splay tree for the same reasons that we have discussed in the previous experiment. In our experiments, it turns out that 50% of the memory allocated to the Splay tree is sufficient to lead to excellent performance.

6 Conclusion

It appears from the preceding discussions and experimentations that the GHOST scheme for index buffering presents a more efficient and robust alternative to other known strategies. The power of our proposed approach centers on the observation that we can store a direct pointer to leaf nodes of a B+-tree without having to buffer the search path of the structure. As a first cut, we have picked the ILRU and Splay tree in our implementation. However, any other buffering mechanisms and data structures having similar properties would have sufficed.

We have experimented with the GHOST scheme under a variety of different workloads, but it has emerged being the best performer in most of the cases. This performance gain is beyond what we have imagined and provides an illustration of the power of the proposed scheme.

Like any deserving piece of work, we are interested in understanding more of its features if not for the time constraints faced. For example, our results show that the optimal allocation of buffers to the Splay tree depends on the workload. This may call for an adaptive approach in dynamic environments where the workload changes. We are currently investigating how to realize such a scheme.

Last but not least, there are a number of policies which we have adopted because of their simplicity. For example, a leaf node of the Splay tree is selected at random for eviction whenever we are out of memory space. There could have been other policies: e.g., pick

the node furthest away from the root, or perform a bulk eviction rather than repeating this operation all so often. All of these options are currently being explored and studied empirically.

Acknowledgement

This work is partially supported by University Research Grant RP982694.

References

[1] G.M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics*, 3:1259–1263, 1962.

[2] R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[3] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. of the ACM SIGMOD Conference*, pages 42–153, 1998.

[4] Chee Yong Chan, Beng Chin Ooi, and Hongjun Lu. Extensible buffer management of indexes. In *Proc. of the 18th VLDB Conference*, pages 444–454, Vancouver, British Columbia, Canada, 1992.

[5] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. of the 11th VLDB Conference*, pages 127–141, 1985.

[6] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[7] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, Dec 1984.

[8] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual IEEE Symposium on Foundations of Coputer Science*, pages 8–21, 1978.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Conference*, pages 47–57, 1984.

[10] Elizabeth J. O'Neil, Patrick E. O'Neil', and Gerhard Weikum. The LRU-K page replacement algorithm for database disk bufferin. In *Proc. of the ACM SIGMOD Conference*, pages 297–306, 1993.

[11] Beng Chin Ooi, Cheng Hian Goh, and Kian-Lee Tan. Fast high-dimensional data search in incomplete databases. In *Proc. of the 24th VLDB Conference*, pages 357–367, New York City, NY, 1998.

[12] Giovanni Maria Sacco. Index access with a finite buffer. In *Proc. of the VLDB Conference*, pages 301–309, 1987.

[13] Giovanni Mario Sacco and Mario Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hotset model. In *Proc. of the 8th VLDB Conference*, pages 257–262, 1982.

[14] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[15] S. B. Yao. Approximating block accesses to database organizations. *Communications of the ACM*, 20:260–261, 1977.