

# Workload Aware B+ Trees

Zhihui Wang      Ke Yi

(Project Report)

## Abstract

Standard B+ trees provide equal access time to all the stored keys, while real world workload is often highly biased, or exhibit strong locality in the access sequences. Previously, DBMSs use buffer management to exploit those characteristics to decrease the average access cost. In this project, we propose a different approach. We reorganize the B+ tree according to previous accesses, trying to capture the locality behavior and decrease the average access cost. We show by experiments that our algorithm can reduce the cost by a reasonable amount in most cases. We also point out that our algorithm can be combined with some buffer management strategy to get even better results.

## 1 Introduction

Indexes have been widely used to speed up the access to the objects in databases. However, for very large database, it is possible that the index itself is so large that only a small fraction of the index can be held in memory. Furthermore, traditional indexes, such as B+ trees, provide equal time to access all the indexed objects. However, the chance to access the objects in database is not uniform[4]. Generally speaking, in a small time frame, a small fraction of all the indexed objects are very possible to get higher chance to be accessed than others. Say, 90% of the queries will access 10% of the total indexed objects. Thus, sometimes it will make sense to speed up the frequent access to the 10% indexed objects at the price of slowing down the access to the other 90% objects.

At present, commercial DBMSs use buffer management to capture this locality of access patterns [3]. In this project, we propose a different approach. We try to reorganize the B+ tree according to previous accesses. We call such a kind of structures *workload-aware B+ trees*.

## 2 Related Work

B+ trees are extensively used as index structures for large databases. It provides optimal worst case search time for all keys stored in the tree. However, when the access distribution is highly skew, the average search time is not optimal any more. A large amount of research is done to improve the average performance of B+ tree indexes. Most of previous research is focused on the buffer management strategy. While pointing out classical buffer management strategies are not suitable for relational database environment [12], new buffer replacement algorithms, e.g. [3], are proposed to take the advantage of access patterns and the special structure of B+ trees. Another similar line of research is to make B+ trees cache conscious in main memory [10].

We notice that the problem of optimizing the average access cost in a B+ tree is a generalized version of the *optimal binary search tree* problem. The best known algorithm uses  $\Theta(n^2)$  time and has the same space complexity [8]. This is not realistic in practice for trees with a large number of keys. Besides, this algorithm is static, in the sense that it depends on the prior knowledge of the access frequencies, sometimes called the *reference probability vector* in the literature. Due to the high complexity of this optimum algorithm, many nearly optimal binary search trees algorithms were proposed, including greedy trees [9], monotonic heuristics, weight balanced trees [1], and some online reorganization algorithms such as [7].

## 3 Problem Definition

In Section 1, we have some intuitive idea of how the problem is. But we still need to define our problem precisely, and on what we are trying to optimize.

A B+ tree is a search tree where all the keys are in the leaf nodes. The out degree of any node is no more than  $B$ , a parameter determined by the size of disk blocks. In theory, the out degrees are also required to be no less than  $\lfloor B/2 \rfloor$  while the root is required to have at least 2 children. This requirement is meant to bound the height of the B+ tree so that the worst-case search time is  $O(\log n)$ . Since our goal is focused on the average case, it does not make sense to retain this requirement. Also, in practice, many implementations of B+ trees simply don't do merging at all and this requirement doesn't hold, either [6]. In our project, we only require that each node has at least two while no more than  $B$  children, except the leaves, who have no children.

Our problem is now the following: starting from an initial B+ tree, we are given a sequence of requests to the keys stored in the B+ tree, one at a time. Because B+ trees are designed mainly for external memory, we only count the number of I/Os as the cost of search, or reorganization. We further assume that the root and all its children are buffered in memory, thus the cost of a request is the number of nodes from root to the leaf containing the key minus 2. At any given time step  $t$ , the algorithm knows all the requests before and at  $t$ , but future requests are hidden from the algorithm. The algorithm then tries to reorganize the B+ tree, without violating the requirement above, to minimize the total cost incurred during a certain period of time. The total cost consists of two parts: the search cost (reading blocks from disks) and the reorganization cost (include maybe both reading and writing the disk blocks). We expect that the total cost will be less than the total cost of the initial static B+ tree without reorganizations, when the requests are highly biased.

To construct the initial B+ tree that we start with, we assume that each node is  $\ln 2 \approx 0.69$  full [13], a reasonable assumption when the tree is the result of a number of random insertions and deletions.

## 4 The Algorithm

### 4.1 Algorithm Overview

Our algorithm is motivated by the idea of weight balanced trees. A weight balanced binary tree is constructed in a top-down manner. The root node is chosen such that the sums of the weights of the subtrees are equal, or as close as possible to being equal, and then the subtrees are built using the same method recursively. Bayer shows in [1] that a weight balanced binary tree is near optimum, by a factor of  $\ln \ln n$ , which is acceptable.

We generalize this idea in the context of a B+ tree. We first assign some kind of weights to each leaf (a block of keys), and the weight of a subtree is defined to be the sum of all the leaves' weights in the subtree. The algorithm dynamically reorganizes the B+ tree. We move a subtree  $T$  up from its parent node to its grandparent if the weight of the  $T$  is no less than the average weight of all the grandparent's subtrees after  $T$  is moved up. The definition and calculation of the weights, as well as the details of the moving-up procedure are described in the following sections.

## 4.2 Weights on Nodes

In order to capture the locality behavior and ease the computation of the weights, we propose the following weight function  $w(t, x)$  on node  $x$  at time  $t$ . When  $x$  is a leaf,  $w(t, x)$  is defined as

$$w(t, x) = \sum_{i=0}^t b(i, x)\theta^{t-i}, \quad (1)$$

where  $b(i, x) = 1$  if  $x$  is visited at time  $i$ ; 0 otherwise. When  $x$  is an internal node,  $w(t, x)$  is defined to be the sum of all the leaves' weights in the subtree rooted at  $x$ .

This definition essentially assigns exponentially decreasing weights to earlier requests of the key. The constant  $\theta$  is a tunable parameter, which should be from 0 to 1. The smaller  $\theta$  is, the more emphasis is put on more recent requests, thus our algorithm becomes more sensitive to the locality in the request sequence. When  $\theta = 1$ , the function basically “remembers” all the previous requests.

The weight function defined in (1) has a nice property: it can be incrementally maintained at little extra cost. Besides the weight itself, we store a *time stamp*  $\tau(x)$  at each node, representing the time when the weight is last computed, i.e., a pair  $(\tau(x), w(\tau(x), x))$  is stored for each node. Then at any later time  $t$ , the updated weight can be computed as

$$w(t, x) = w(\tau(x), x)\theta^{t-\tau(x)}, \quad (2)$$

if  $x$  is not visited after  $\tau(x)$ . For each access, we update all the nodes along the path from root to the leaf as

$$w(t, x) = w(\tau(x), x)\theta^{t-\tau(x)} + 1. \quad (3)$$

It's not hard to show that with (2) and (3), we can always keep the weights updated. We assume that all these statistics information are kept in main memory, then the cost of the incremental maintenance of these statistics can be ignored. In the following, we omit the time parameter  $t$  in  $w(t, x)$ , when there is no ambiguity.

## 4.3 Reorganization

As described in Section 4.1, we move up a subtree  $T$  when  $w(t, T)$  is no less than the average weight of its “dad” and all its “uncles”. To be more

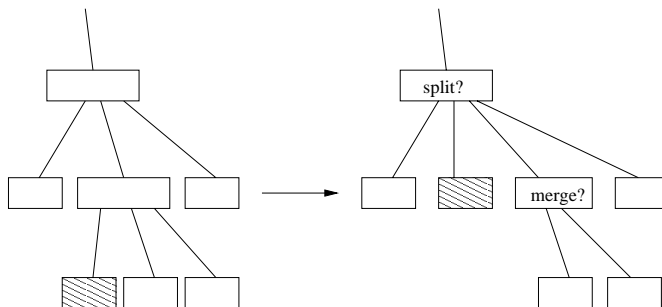


Figure 1: When  $x$  is at one end.

precise, after an request is processed, for each (child, grandparent) pair  $(x, g(x))$  along the access path, we compute an ratio as

$$r(x) = \begin{cases} \frac{w(x)}{w(g(x))/(d(g(x))+1)}, & \text{if } d(g(x)) < B; \\ \frac{w(x)}{w(g(x))/(d(g(x))/2+1)}, & \text{if } d(g(x)) = B, \end{cases}$$

where  $d(x)$  is the out degree of  $x$ . For all the  $x$ 's with  $r(x) > 1$ , we select out the the one with the large ratio and perform a move-up operation on it. Note that the definition of  $r(x)$  above has taken into account the possible splitting of  $g(x)$  when  $x$  is moved up.

There are two cases of the move-up operation depending on the location of  $x$ . The simpler one is shown in Figure 1. The shaded box is  $x$  that is to be moved up. After we change the pointers, we need to do a MERGE operation on  $x$ 's old parent and then a SPLIT on  $x$ 's new parent, if necessary. The MERGE and SPLIT operations are the same as those in the standard B+ tree algorithm. We do a MERGE on  $x$  when  $d(x) < B/2$ . First, if one of  $x$ 's siblings, say  $y$ , has enough children and  $d(x) + d(y) \geq B$ , we then drag a portion of  $y$ 's children to  $x$  such that both  $x$  and  $y$  have no less than  $B/2$  children. If none of  $x$ 's siblings has enough children, we merge  $x$  with one of them to form a larger node. A special case in doing a MERGE is when  $d(x) = 1$ . In this case, we simply remove  $x$  and let  $x$ 's child replace its position. If  $d(x) > B$ , we have to do a SPLIT, producing two separate nodes. Note that SPLIT is recursive, that is, after  $x$  is splitted, we also need to do a SPLIT on  $x$ 's parent. [6] contains a more detailed description on these operations.

The second and more general case is when  $x$  is in the middle among its siblings, shown in Figure 2. In this case, after changing the pointers, we need to consider doing MERGE on two nodes and then SPLIT on one node. Please

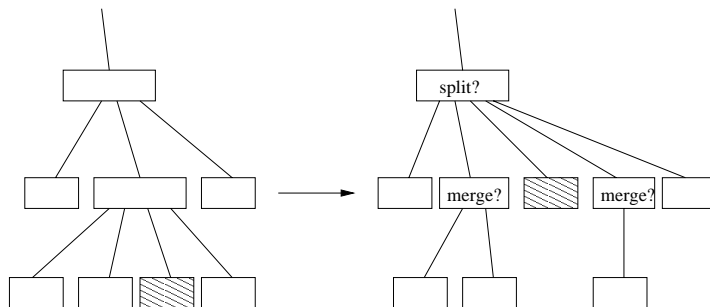


Figure 2: When  $x$  is in the middle.

note that the order of the MERGE and SPLIT operations are important. We need to do MERGES first since this may save unnecessary SPLITS.

The operations introduced above ensure that the out degree of any node is always no more than  $B$ . However, a small number of nodes may have less than  $B/2$  children. We have already justified this problem in Section 3. We will further show by experiments that this will not cause the performance to degrade, nor produce too many nodes in the B+ tree.

## 5 Experiments

### 5.1 Methodology

We have implemented the B+ tree structure and our reorganization algorithm in main memory to simulate the extern memory scenario. The initial B+ tree is constructed using bulk-loading, but each node has only  $\ln 2 \cdot B$  children [13]. The maximum fanout of a node in the B+ tree and the total number of keys stored in the B+ tree are the tunable parameters in our implementation. These two parameters indicate the size and depth of original B+ tree. The maximum fanout of a node also indicates the size of each node in the B+ tree to some degree.

We have generated two kinds of request sequences as our test datasets, which follow Zipf-like distributions and time-dependent distributions respectively. For the datasets following Zipf-like distribution, we purposely do not consider the query patterns vary with time, and just intent to demonstrate how our algorithm capture the skew characteristic of the query sequences. We exploit how our algorithm works well with the time-related query sequences by using the datasets following time-dependent distribution. In the experiments on Zipf-like query sequences, we construct an initial B+ tree

that stores 1000 keys, and has a maximum fanout of 6 in each node. In the experiments on time-dependent query sequences, we have an initial B+ tree with 2000 keys and a maximum fanout of 12. These parameters produce B+ trees with heights 4 or 5, which is meant to “scale down” typical B+ trees to ones that are suitable in our experiments.

We believe that the disk I/Os will be the dominating factor in the performance of our algorithm. So We carefully count the number of disk I/Os needed in the search and reorganization operations of our algorithm, and show that the number of disk I/Os required in our algorithm is less than that in traditional B+ tree index in the case of both Zipf-like request distributions and time-dependent distributions.

## 5.2 Zipf-like Distributions

Zipf-like distribution is one of the typic skew distributions. Typical Zipf distributions [14] follow the formula  $p_i \sim 1/i^\alpha$ , ( $0 < \alpha < 1$ ). It means that the probability  $p_i$  of a request to the  $i$ th most popular object is proportional to  $1/i^\alpha$ . Large  $\alpha$ 's lead to more biased distributions; and small  $\alpha$ 's correspond to the long, heavy tails in the distributions. In our experiments, we first randomly assign a popularity number to each key stored in the B+ tree, then the requests are generated according to probabilities  $1/i^\alpha$  after normalization, where  $i$  is the assigned popularity. Figure 3 is an example of our synthetic Zipf-like query sequences, in which Zipf's  $\alpha = 0.8$ . The  $x$  axis is the sequence number of queries, and the  $y$  axis is the value of requested keys. Each point in this figure represents a query.

In our simulations, we synthesize some Zipf-like query sequences by varying the value of  $\alpha$  from 0.5 to 0.95, with 65000 requests in each of these synthetic distributions. In this experiment, we are using biased distributions with no (or little) locality. Intuitively, since all requests are generated from the same distribution, each previously access object is an indicator of how the future requests should be. Our algorithm should “remember” all the history, to achieve the best results. Our experiment confirms this intuition: the performance is the best when  $\theta = 1$ , and degrades as  $\theta$  gets smaller. At certain points, the performance is even worse than the static B+ tree when the cost of reorganization is so large that it cannot offset the gain from reduces search cost. For this reason, we only show the result for  $\theta = 1$ . This can be generalized to the following: if the requests are produced from a static distribution,  $\theta = 1$  is always the best choice.

Figure 4 shows the speedup that our workload-aware index can get in the case of the query sequences following Zipf-like distributions. The  $x$

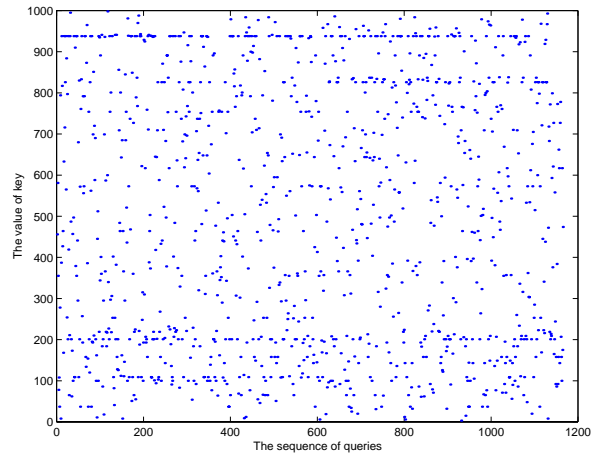


Figure 3: The query sequence following Zipf-like distribution.

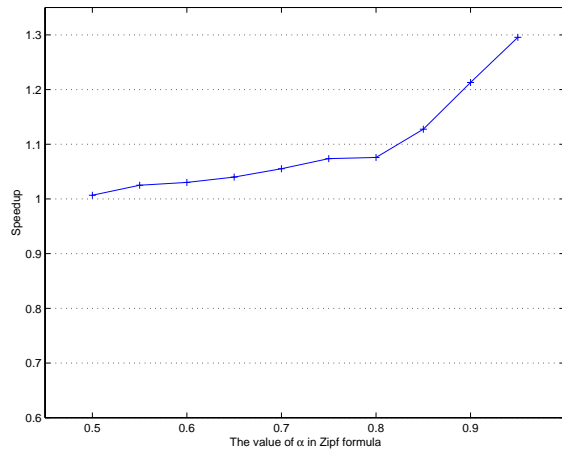


Figure 4: The speedup in the Zipf-like query sequences.

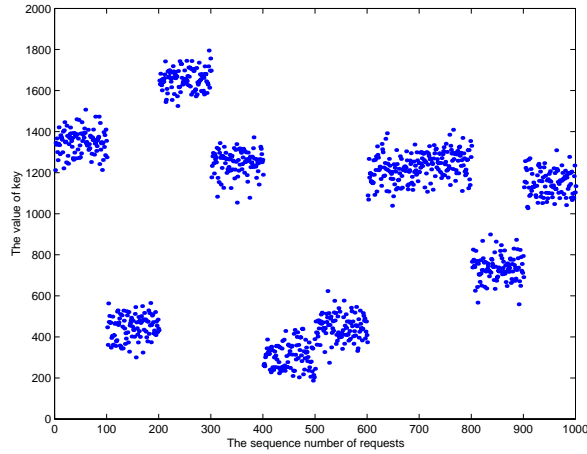


Figure 5: The query sequence following time-dependent distribution.

axis is the value of  $\alpha$  in Zipf formula. The  $y$  axis is the speedup, which is defined as the ratio of the number of disk I/Os in the traditional B+ index algorithm (only including searching operations) to that in our algorithm (including reorganization and searching operations). Each point in the plot is the average of 5 repetitions of the experiments, using different random seeds. It shows we can get the an average speedup of 1.1 when  $\alpha$  varies from 0.5 to 0.95, and our algorithm can get better performance for more skew query distribution. For instance, when  $\alpha = 0.9$ , we can get a speedup by more than 1.2.

### 5.3 Time-dependent Distributions

In this section, we illustrate how our algorithm works well in the case of the time-dependent requests sequences, exhibiting strong locality.

An example of our synthetic time-dependent query sequences is shown in Figure 5. The  $x$  axis is the sequence number of the queries, and the  $y$  axis is the *key* of objects. To generate such distributions, we first divide the whole duration into a number of *time-steps* of equal length. Within each time-step, we generate requests according a normal distribution, with a uniformly random mean and a fixed variance of 20. This is meant to capture the shifts in people’s interests: there are different popular objects (that are close to each other) at different times. This also corresponds to the concept of *working set*, suggested in many situations.

Figure 6 is the speedup of our workload-aware B+ tree in the case of

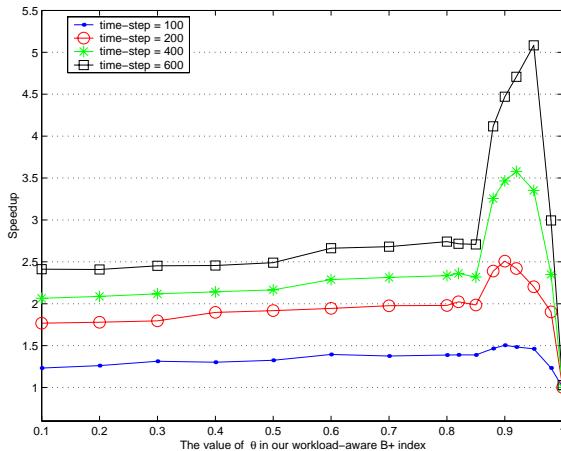


Figure 6: The speedup in the time-dependent query sequences.

different values of  $\theta$  in our algorithm and different time-steps in the query sequences. The definition of speedup is same as the previous section. In our experiments, we have a B+ trees with 2000 keys and a maximum fanout of 12 in all these experiments. The total number of requests in our experiments is 10000. Again, each point in the plot is the average of 5 independent repetitions of the experiments.

As can be seen, our algorithm can get good speedup in most cases. Particularly, for all the time-steps, it gets better speedup when  $\theta \geq 0.85$ , and reaches its maximum speedup when  $\theta$  between 0.9 and 0.95. At the extreme case that  $\theta = 1$ , there is nearly no reorganization, and our algorithm degrades to static B+ tree index. Furthermore, we find that the larger the time-step is, the higher speedup our algorithm can get. When the time-step is 600, we even reach the speedup of more than 5 for  $\theta = 0.95$ . Generally speaking, our algorithm can adapt well to the time-dependent query sequences with some appropriate value of  $\theta$  in our algorithm. At the present, the value of  $\theta$  in our algorithm is a tunable parameter specified at the beginning of the algorithm. In the further work, we will expect to have the value of  $\theta$  adaptive to the various time-dependent query patterns automatically.

As pointed out earlier, the number of nodes in our workload-aware B+ index may increase with the running of our algorithm, since there is SPLIT operations resulting from the moving-up of the popular nodes. One may worry that the our B+ tree might use too much space to trade for reduced search cost. We here show that this does happen in practice. Figure 7 shows three traces for the number of nodes in our workload-aware B+ tree index.

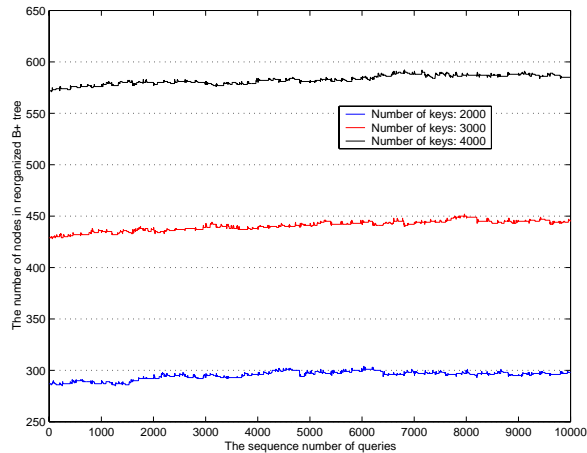


Figure 7: The traces for the number of nodes in workload-aware B+ index

The  $x$  axis represents the sequence number of the queries, and the  $y$  axis represents the corresponding number of nodes in our reorganized B+ tree index. There are 3 traces shown in the Figure, which corresponds to the B+ tree with 2000, 3000, and 4000 keys respectively. In our experiments, each node of the B+ trees has the maximum fanout of 12, the value of  $\theta$  in our algorithm is 0.9, the time-steps in all the sequences are 200, and the number of the requests is 10000. As can be seen, the number of nodes in our workload-aware B+ tree index increases slightly at the beginning, and then keeps stable at some non-large value. Furthermore, the traces show that the increase is independent of the number of nodes in the initial B+ tree. We have also conducted the experiments with different values of time-step, and for Zipf like distributions, their traces is similar to those in Figure 7.

## 6 Future Work

We have done a initial study about the workload-aware B+ index. However, there are some limitations in our present work, which we will further explore in our future work.

First, we do not consider insertions and deletions. But we believe that these operations under our workload-aware index are similar to those under traditional B+ trees. It will be easy to incorporate them into our framework.

Second, we have only deployed a simple buffer management in our experiments, caching the upper two level of B+ index tree. The study of Doyle,

Chase et. al. in [5] shows that the caches or buffers have the trickle-down effect. Our initial study shows that our workload-aware B+ index can further capture the moderately popular queries in the miss stream of buffers or caches. We will further explore the scenario by deploying more delicate buffer management or caching strategies[2] in our future study. We expect that the combined strategy will outperform simple buffer management alone.

Third, our present algorithm intent to logically reorganize the B+ index adaptive to the workload. It is also possible to physically reorganize the B+ index according to the workload. For example, physically moving the index nodes close to the most popular objects will improve the locality and get better performance, and aligning the popular index nodes into the same disk track will avoid the rotational latency[11]. We will further explore the benefit of building workload-aware B+ index by combining the physical characteristics of disks.

Finally, a limitation in our algorithm is that the parameter  $\theta$  is somehow empirical. For some  $\theta$ , the performance might be even worse than the static B+ trees. How to automatically and adaptively tune this parameter remains to be future work.

## 7 Conclusion

In this paper, we make an initial study on workload-aware B+ trees. Our approach is to reorganize the B+ index via moving up popular objects and accelerating the access to them, thus reaching better average performance. We first design a weight function to indicate the importance of each nodes. With efficient incremental maintenance of these weight information, we reorganize the B+ tree, using a heuristic motivated from nearly optimal binary trees, to shorten the depths of important nodes.

We have done some initial experiments with synthetic request sequences following Zipf-like distributions and time-dependent distributions. For Zipf-like request sequences, our algorithm can get speedups up to 1.3. For time-dependent distributions with strong locality, our algorithm can achieve speedups up to 5. In addition, we also show that our workload-aware B+ tree only brings slight increase in the number of index nodes.

## 8 Acknowledgment

We would like to thank Dr. Jun Yang and Dr. Jeffrey S. Chase for their helpful suggestions.

## References

- [1] P. J. Bayer. Improved bounds on the costs of optimal and balanced search trees. Technical Memo MIT/LCS/TM-69, Massachusetts Institute of Technology, Laboratory for Computer Science, Nov. 1975.
- [2] M. Castro, A. Adya, B. Liskov, and A. C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *Proceedings of the ACM Symposium on Operating System Principles(SOSP'97)*, Saint-Malo, France, Octer 1997.
- [3] C. Y. Chan, B. C. Ooi, and H. Lu. Extensible buffer management of indexes. In L.-Y. Yuan, editor, *18th International Conference on Very Large Data Bases*, pages 444–454, Vancouver, Canada, 23–27 Aug. 1992. Morgan Kaufmann.
- [4] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems(TODS)*, 9(2):163–186, 1984.
- [5] R. Doyle, J. Chase, S. Gadde, and A. Vahdat. The trickle-down effect: Web caching and server request distribution. In *Sixth International Workshop on Web Caching and Content Delivery*, June 2001.
- [6] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems: the complete book*. Prentice Hall, 2002.
- [7] Hofri and Shachnai. Efficient reorganization of binary search trees. *ALGRTHMICA: Algorithmica*, 31, 2001.
- [8] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, Jan. 1971.
- [9] J. F. Korsh. Greedy binary search trees are nearly optimal. *Information Processing Letters*, 13(1):16–19, Oct. 1981.
- [10] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(2), 2000.
- [11] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Conference on File and Storage Technologies(FAST)*, Monterey, CA, January 2002.

- [12] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.
- [13] A. Yao. Random 3-2 trees. *Acta Informatica, Springer Verlag (Heidelberg, FRG and NewYork NY, USA) Verlag*, 2(9), 1978.
- [14] G. Zipf. *Human Behavior and the principle of Least Effort*. Addison Wesley, 1949.