Fractal Prefetching B^+-Trees: Optimizing Both Cache and Disk Performance

Shimin Chen   Phillip B. Gibbons[†]   Todd C. Mowry
Gary Valentin[‡]

March 2002
CMU-CS-02-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Information Sciences Research Center, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.
Current affiliation: Intel Research Pittsburgh, 417 South Craig Street, Pittsburgh, PA 15213.
[‡]IBM Toronto Lab, 8200 Warden Avenue, Markham, Ontario, L6G 1C7, Canada.

Abstract

B^+-Trees have been traditionally optimized for I/O performance with disk pages as tree nodes. Recently, researchers have proposed new types of B^+-Trees optimized for CPU cache performance in main memory environments, where the tree node sizes are one or a few cache lines. Unfortunately, due primarily to this large discrepancy in optimal node sizes, existing disk-optimized B^+-Trees suffer from poor cache performance while cache-optimized B^+-Trees exhibit poor disk performance. In this paper, we propose *fractal prefetching B^+-Trees* (fpB^+-Trees), which embed "cache-optimized" trees within "disk-optimized" trees, in order to optimize both cache and I/O performance. We design and evaluate two approaches to breaking disk pages into cache-optimized nodes: *disk-first* and *cache-first*. These approaches are somewhat biased in favor of maximizing disk and cache performance, respectively, as demonstrated by our results. Both implementations of fpB^+-Trees achieve dramatically better cache performance than disk-optimized B^+-Trees: a factor of 1.1–1.8 improvement for search, up to a factor of 4.2 improvement for range scans, and up to a 20-fold improvement for updates. In addition, fpB^+-Trees accelerate I/O performance for range scans by using jump-pointer arrays to prefetch leaf pages, thereby achieving a speed-up of 2.5–5 on IBM's DB2 Universal Database.

This research is supported in part by grants from the National Science Foundation, Microsoft, and Intel.

Keywords: Cache performance, I/O performance, prefetching, databases, B^+-Tree index

1 Introduction

The B^+-Tree is a ubiquitous structure for indexing disk-resident data. It provides basic index operations such as search, range scan, insertion and deletion, while minimizing the number of disk accesses. To optimize I/O performance, traditional "disk-optimized" B^+-Trees are composed of nodes the size of a *disk page*—i.e., the natural transfer size for reading or writing to disk. Recently, several studies [5, 6, 19] have considered B^+-Tree variants for indexing memory-resident data. These studies present new types of B^+-Trees—*cache-sensitive B^+-Trees* [19], *partial-key B^+-Trees* [5], and *prefetching B^+-Trees* [6]—that optimize for CPU cache performance by minimizing the impact of cache misses. These "cache-optimized" B^+-Trees are composed of nodes the size of a *cache line*[1]—i.e., the natural transfer size for reading or writing to main memory.

Unfortunately, B^+-Trees optimized for disk suffer from poor CPU cache performance, and B^+-Trees optimized for cache suffer from poor I/O performance. This is primarily because of the large discrepancy in node sizes: disk pages are typically 4KB–64KB while cache lines are often 32B–128B, depending on the system. Thus existing disk-optimized B^+-Trees suffer an excessive number of cache misses to search in a (large) node, wasting time and forcing the eviction of useful data from the cache. Likewise, existing cache-optimized B^+-Trees, in searching from the root to the desired leaf, may fetch a distinct page for each node on this path. This is a significant performance penalty, for the smaller nodes of cache-optimized B^+-Trees imply much deeper trees than in the disk-optimized cases (e.g., twice as deep). The I/O penalty for range scans on nonclustered indexes of cache-optimized trees is even worse: a distinct page may be fetched for each leaf node in the range, increasing the number of disk accesses by the ratio of the node sizes (e.g., a factor of 500).

1.1 Our Approach: Fractal Prefetching B^+-Trees

In this paper, we propose and evaluate *Fractal Prefetching B^+-Trees* (fpB^+-Trees), which are a new type of B^+-Tree that optimizes *both* cache and I/O performance. In a nutshell, an fpB^+-Tree is a single index structure that can be viewed at two different granularities: at a coarse granularity, it contains disk-optimized nodes that are roughly the size of a disk page, and at a fine granularity, it contains cache-optimized nodes that are roughly the size of a cache line. We refer to a fpB^+-Tree as being "fractal" because of its self-similar "tree within a tree" structure, as illustrated in Figure 1. The cache-optimized aspect is modeled after the *prefetching B^+-Trees* that we proposed earlier [6], which were shown to have the best main memory performance for fixed-size keys. (We note, however, that this general approach can be applied to any cache-optimized B^+-Tree.) In a prefetching B^+-Tree, nodes are several cache lines wide (e.g., 8—the exact number is tuned according to various memory system parameters), and prefetching is used so that the time to fetch a node is not much longer than the delay for a single cache miss.

We design and evaluate two approaches to implementing fpB^+-Trees: (i) *disk-first* and (ii) *cache-first*. In the *disk-first* approach, we start with a disk-optimized B^+-Tree, but then organize the keys and pointers within each page-sized node as a small tree. This *in-page tree* is a variant of the prefetching B^+-Tree. To pack more keys and pointers into an in-page tree, we use short in-page offsets rather than full pointers in all but the leaf nodes of an in-page tree. We also show the advantages of using different sizes for leaf versus non-leaf nodes in an in-page tree. In contrast, the *cache-first* approach starts with a cache-optimized prefetching B^+-Tree (ignoring disk page boundaries), and then attempts to group together these smaller nodes into page-sized nodes to optimize disk performance. Specifically, the cache-first approach seeks to place a parent and its children on the same page, and to place adjacent leaf nodes on the same page. Maintaining both structures as new keys are added and nodes split poses particular challenges. We will show how to process insertions and deletions efficiently in both disk-first and cache-first fpB^+-Trees. We select the optimal node sizes in both approaches to maximize the number of entry slots in a leaf page while analytically achieving search cache performance within 10% of the best.

Ideally, both the *disk-first* and the *cache-first* approaches would achieve identical data layouts, and hence equivalent cache and I/O performance. In practice, however, the mismatch that almost always occurs between the size of a cache-optimized subtree and the size of a disk page (in addition to other implementation details such as full pointers versus page offsets) causes the disk-first and cache-first approaches to be slightly biased

[1] In the case of *prefetching B^+-Trees* [6], the nodes are several cache lines wide.

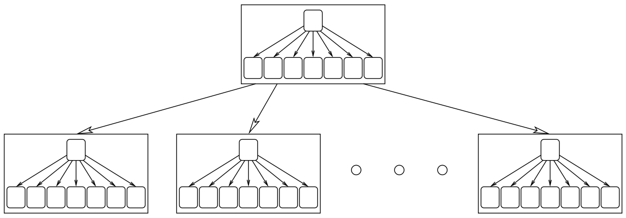


Figure 1: Self-similar “tree within a tree” structure

in favor of disk and cache performance, respectively. Despite these slight disparities, both implementations of fpB$^+$-Trees achieve dramatically better cache performance than disk-optimized B$^+$-Trees.

To accelerate range scans, fpB$^+$-Trees employ the *jump-pointer array* scheme that we proposed earlier [6]. A jump-pointer array contains the leaf node addresses of a tree, which are used in range scans to prefetch the leaf nodes, thus speeding up the scans. In [6], we showed that this approach significantly improves cache performance. In this paper, we show it is also beneficial for I/O, by demonstrating a factor of 2.5–5 improvement in the range scan I/O performance for IBM's DB2 running on a multi-disk platform.

1.2 Related Work

A number of recent studies have demonstrated the importance of optimizing the cache performance of a DBMS [1, 2, 3]. B$^+$-Trees have been discussed in this regard, including several recent survey papers [11, 16]. This paper, however, is the first to propose a B$^+$-Tree index structure that effectively optimizes both CPU cache and disk performance on modern processors, for each of the basic B$^+$-Tree operations: searches, range scans, insertions, and deletions.

Chilimbi *et al.* [8] demonstrated that B$^+$-Trees with cache line sized nodes can outperform binary trees for memory-resident data on modern processors. Likewise, B$^+$-Trees outperform T-trees [14] on today's processors [18]. Lomet [15] presented techniques for selecting an optimal B$^+$-Tree page size when considering buffer cache performance, for disk-resident data. Lomet's recent survey of B$^+$-Tree techniques [16] mentioned the idea of intra-node micro-indexing: i.e., placing a small array in a few cache lines of the page that indexes the remaining keys in the page. While it appears that this idea had not been pursued in any detail before, we compare its performance against fpB$^+$-Trees later in our experimental results. We observe that while micro-indexing achieves good search performance (often comparable to fpB$^+$-Trees), it suffers from poor update performance. As part of future directions, Lomet [16] has independently advocated breaking up B$^+$-Tree disk pages into cache-friendly units, pointing out the challenges of finding an organization that strikes a good balance between search and insertion performance, storage utilization, and simplicity. We believe that fpB$^+$-Trees achieve this balance. Graefe and Larson [11] presented a survey of techniques for improving the CPU cache performance of B$^+$-Tree indexes. They discussed a number of techniques, such as key compression, that are complementary to our study, and could be incorporated into fpB$^+$-Trees. Bender *et al.* [4] present a recursive B$^+$-Tree structure that is *asymptotically* optimal, regardless of the cache line sizes and disk page sizes, but assuming no prefetching.

1.3 Contributions of This Paper

This paper makes the following contributions. First, we propose and evaluate *Fractal Prefetching B$^+$-Trees* (fpB$^+$-Trees) as a novel index structure that optimizes both cache and disk performance simultaneously. Second, we present detailed analysis of the fundamental tradeoffs between the *disk-first* and the *cache-first* implementations of fpB$^+$-Trees. While the performance of each of these implementations remains slightly biased toward its original goal, both versions of fpB$^+$-Trees improve upon the cache performance of disk-optimized B$^+$-Trees (without significantly degrading I/O performance) as follows: (i) a factor of 1.1–1.8 improvement for search; (ii) up to a factor of 4.2 improvement for range scans; and (iii) up to a 20-fold improvement for updates. Third, we present the first detailed evaluation of *micro-indexing* [16], and find that its poor update performance makes it less attractive than fpB$^+$-Trees. Finally, we demonstrate that fpB$^+$-Trees can also be used to accelerate *I/O performance*. In particular, we demonstrate an over twofold to fivefold improvement for index range scans in an industrial-strength commercial DBMS (IBM's DB2).

The remainder of this paper is organized as follows. Section 2 describes how fpB$^+$-Trees enhance I/O performance. Then Section 3 describes how they enhance cache performance while preserving I/O performance. Section 4 presents experimental results validating the effectiveness of fpB$^+$-Trees in optimizing both cache and disk performance. Section 5 discusses several related problems and ideas. Section 6 presents our conclusions.

2 Optimizing I/O Performance

Fractal Prefetching B$^+$-Trees combine features of disk-optimized B$^+$-Trees and cache-optimized B$^+$-Trees to achieve the best of both structures. In this section, we describe how fpB$^+$-Trees improve I/O performance for modern database servers. In a nutshell, we consider applying to disk-resident data each of the techniques in [6] for improving the cache performance for memory-resident data. We argue that while the techniques are not advantageous for search I/O performance, they can significantly improve range scan I/O performance.

Modern database servers are composed of multiple disks per processor. For example, many TPC benchmark reports are for SMP servers with 10-30 disks per processor, and hundreds of disks in all. To help exploit this raw I/O parallelism, commercial database buffer managers use techniques such as sequential I/O prefetching and delayed write-back. While sequential I/O prefetching helps accelerate range scans on *clustered* indexes, it offers little or no benefit for range scans on *non-clustered* indexes or for searches. Our goal is to effectively exploit I/O parallelism by explicitly prefetching disk pages even when the access patterns are not sequential.

In a previous paper [6], we proposed and evaluated *prefetching B$^+$-Trees* (pB$^+$-Trees) as a technique for enhancing CPU cache performance for index searches and index range scans on memory-resident data. The question that we address now is whether those same techniques can be applied to accelerating I/O performance for disk-resident data. Since the relationship between main memory and disk for a disk-optimized tree is somewhat analogous to the relationship between CPU cache and main memory for a cache-optimized tree, one might reasonably expect the benefit of a technique to translate in at least some form across these different granularities [11]. However, because of the significant differences between these two granularities (e.g., disks are larger and slower, main memory is better suited to random access, etc.), we must carefully examine the actual effectiveness of a technique at a different granularity. In Sections 2.1 and 2.2, we consider the two aspects of pB$^+$-Trees which accelerate *searches* and *range scans*, respectively.

2.1 Searches: Prefetching and Node Sizes

To accelerate search performance, our pB$^+$-Tree design [6] increased the size of a B$^+$-Tree node size to be multiple cache lines wide and prefetched all cache lines within a node before accessing it. In this way, the multiple cache misses of a single node are serviced in parallel, thereby resulting in an overall miss penalty that is only slightly larger than that of a single cache miss. The net result is that searches become faster because nodes are larger and hence trees are shallower.

For disk-resident data, the page-granularity counterpart is to increase the B$^+$-Tree node size to be a multiple of the disk page size and prefetch all pages of a node when accessing it. By placing the pages that make up a node on different disks, the multiple page requests can be serviced in parallel. For example, a 64KB node could be striped across 4 disks with 16KB page size, and read in parallel. As in the cache scenario, faster searches may result.

However, there are drawbacks to applying this approach to disks. While the I/O latency is likely to improve for a single search, the I/O throughput may become worse because of the extra seeks for a node. In an OLTP environment, multiple transactions can overlap their disk accesses, and the I/O throughput is often dominated by seek times; hence additional seeks may degrade performance. Note that this is not a problem for CPU cache performance since only the currently executing thread can exploit its cache hierarchy bandwidth.

In a DSS environment, a server is often dedicated to a single query at a time, and hence latency determines throughput. Thus multipage-sized nodes spanning multiple disks may improve search performance. However,

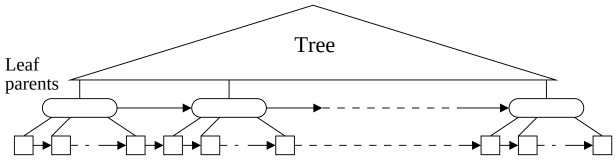


Figure 2: Internal jump-pointer array

search times may be less important to overall DSS query times, which are often dominated by operations such as range scans, hash joins, etc. Moreover, "random" searches are often deliberately avoided by the optimizer. An indexed nested loop join may be performed by first sorting the outer relation on the join key [13, 10]. Thus each key lookup in the inner relation is usually adjacent to the last lookup, leading to an I/O access pattern that essentially traverses the tree leaf nodes in order (similar to range scans).

For these reasons, we do not advocate using multipage-sized nodes. Hence throughout this paper, our target node size for optimizing the disk performance of fpB$^+$-Trees will be a *single* disk page.

2.2 Range Scans: Prefetching via Jump-Pointer Arrays

For range scan performance, our previous paper [6] proposed a *jump-pointer array* structure that permits the leaves in the range scan to be effectively prefetched. A range scan is performed by searching for the starting key of the range, then reading consecutive leaf nodes in the tree (following the sibling links between the leaf nodes) until the end key for the range is encountered. One implementation of the jump-pointer array is shown in Figure 2: An *internal* jump-pointer array is obtained by adding sibling pointers to each node that is a parent of leaves. These leaf parents collectively contain the addresses for all leaf nodes, facilitating leaf node prefetching. By issuing a prefetch for each leaf node sufficiently far ahead of when the range scan needs the node, the cache misses for these leaves are overlapped.

The same technique can be applied at page granularity to improve range scan I/O performance, by overlapping leaf page misses. It is particularly helpful in non-clustered indexes and when leaf pages are *not* sequential on disks, a common scenario for frequently updated indexes.[2] Note that the original technique [6] prefetched past the end key. This overshooting is not a major concern at cache granularity; however, it can incur a large penalty at page granularity both because each page is more expensive to prefetch and because we must prefetch farther ahead in order to hide the larger disk latencies. To solve this problem, fpB$^+$-Trees begin by searching for both the start key and the end key, remembering the range end page. Then when prefetching using the leaf parents, we can avoid overshooting. Also note that because all the prefetched leaf pages would have also been accessed in a plain range scan, this technique does not decrease throughput.

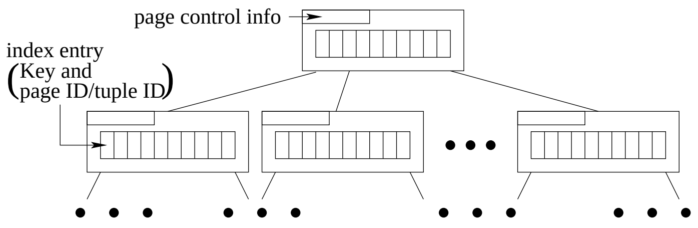
Our previous paper [6] also described an alternative implementation of the jump-pointer array: An *external* jump-pointer array maintains a chunked-linked list data structure external to a B$^+$-Tree, containing leaf node addresses. For most index structures in this paper, we will use internal jump-pointer arrays for ease of implementation. But when the index structure does not allow effective implementation of internal jump-pointer arrays, we will choose external jump-pointer arrays to prefetch for range scans. Please see details in Section 3.3.

The jump-pointer array approach is applicable for improving the I/O performance of standard B$^+$-Trees, not just fractal ones, and as our experimental results will show, can lead to a fivefold or more speedup for large range scans.

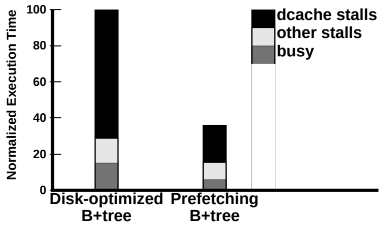
3 Optimizing Cache Performance

In this section, we describe how fpB$^+$-Trees optimize CPU cache performance without sacrificing their I/O performance. Although B$^+$-Trees for disk-resident data have traditionally ignored CPU cache performance because search and update times were dominated by I/O costs, recent studies have demonstrated the importance of CPU cache performance [1, 2, 3]. Most modern database server machines have sufficient disk

[2]For clustered indexes or when leaf pages are sequential on disks, sequential I/O prefetching can be employed instead.



(a) page organization



(b) execution time breakdown for search

Figure 3: Disk-optimized B^+-Trees.

bandwidth such that they are typically not I/O bound, but their processors are stalled a significant fraction of the time while servicing CPU data cache misses.

Why traditional B^+-Trees suffer poor cache performance In a traditional disk-optimized B^+-Trees, each tree node is a page (typically 4KB–64KB). Figure 3(a) depicts a B^+-Tree, assuming fixed length keys.[3] A small part of the page contains page control information. The bulk of the page contains a sorted array of keys, together with either the page ID for its child node (if the node is a nonleaf) or the tuple ID for a tuple (if the node is a leaf). We will refer to a key and either its page ID or tuple ID as an *entry*.

During a search, each page on the path to the key is visited, and a binary search is performed on the very large contiguous array in the page. This binary search is quite costly in terms of cache misses. A simple example helps to illustrate this point. If the key size, page ID size, and tuple ID size are all 4 bytes, an 8KB page can hold over 1000 entries. If the cache line size is 64 bytes, then a cache line can only hold 8 entries. Imagine a certain page has 1023 entries numbered 1 through 1023. To locate a key matching entry 71, a binary search will perform ten probes, for entries 512, 256, 128, 64, 96, 80, 72, 68, 70, and 71, respectively. Assuming that the eight entries from 65 to 72 fall within a single cache line, the first seven probes are all likely to suffer cache misses. The first six of the seven misses are especially wasteful, since each of them brings in a 64B cache line but uses only 4B of that line. Only when the binary search finally gets down to within a cache line are more data in a cache line used. This lack of spatial locality makes binary search on a very large array suffer from poor cache performance.

Figure 3(b) compares the performance of disk-optimized B^+-Trees with cache-optimized prefetching B^+-Trees [6] for searches. The figure shows the simulated execution time (normalized to disk-optimized B^+-Trees) for performing 2000 random searches after each tree has been bulkloaded with 10 million keys on a memory system similar to the Compaq ES40 [9]—details are provided later in Section 4.1. Execution time is broken down into busy time, data cache stalls, and other stalls. As we see in Figure 3(b), disk-optimized B^+-Trees spend significantly more time stalled on data cache misses than prefetching B^+-Trees.[4].

In addition to search, updates are also costly. Insertion and deletion both begin with a search, which has poor cache performance. Another problem is that in order to insert an entry into a sorted array, half of the page (on average) must be copied to make room for the new entry. To make matters worse, the optimal disk page size for B^+-Trees is increasing with disk technology trends [12, 15], making the above problems even more serious in the future.

Techniques for improving B^+-Tree cache performance One approach that was briefly mentioned by Lomet [16] is *micro-indexing*, which is illustrated in Figure 4. The idea behind micro-indexing is that the first key of every cache line in the array can be copied into a smaller array, such as keys $1, 9, 17, \ldots, 1017$ in the example above. These 128 keys are searched first to find the cache line that completes the search (thus reducing the number of cache misses to five in the example). Unfortunately this approach does not address

[3]The issues and solutions for fixed length keys are also important for variable length keys, which have their own added complications in trying to obtain good cache performance [5]. Details are in Section 5.

[4]The extra "busy" time for disk-optimized B^+-Trees is due to the instruction overhead associated with buffer pool management; note that this does not translate into extra data cache stall time due to how we conduct our simulations, as discussed later in Section 4.1

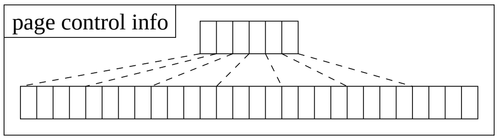


Figure 4: Illustration of micro-indexing

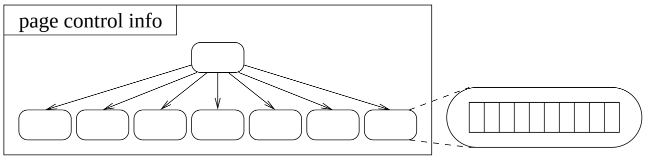


Figure 5: Disk-first fpB$^+$-Tree: a cache-optimized tree inside each page

the data movement problem upon index updates, and therefore it suffers poor update performance (as we will see later in Section 4.2).

To realize good cache performance for all B$^+$-Tree operations, we look to cache-optimized B$^+$-Trees as a model and propose to break disk-sized pages into cache-optimized nodes. This is the guiding principle behind fpB$^+$-Trees. We propose and evaluate two approaches for embedding cache-optimized trees into disk-optimized B$^+$-Tree pages: *disk-first* and *cache-first*. Section 3.1 describes the disk-first approach, while Section 3.2 describes the cache-first approach, both focusing on searches and updates. Then in Section 3.3, we discuss range scans for both approaches.

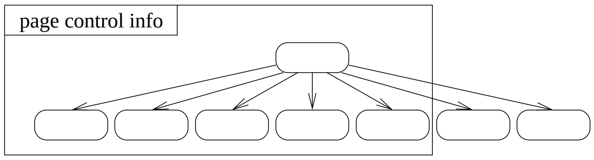
3.1 Disk-First fpB$^+$-Trees

Disk-first fpB$^+$-Trees start with a disk-optimized B$^+$-Tree, but then organize the keys and pointers in each page-sized node into a cache-optimized tree, as shown in Figure 5. The large contiguous array in a traditional disk-optimized B$^+$-Tree page is replaced by a small cache-optimized tree, which we call an *in-page tree*. Our in-page trees are modeled after pB$^+$-Trees, because they were shown to have the best cache performance for memory-resident data with fixed-length keys [6]. The approach, however, can be applied to any cache-optimized tree.

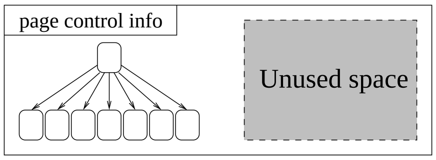
As in a pB$^+$-Tree, an fpB$^+$-Tree in-page tree has nodes that are aligned on cache line boundaries. Each in-page node is several cache lines wide. When an in-page node is to be visited as part of a search, all the cache lines comprising the node are *prefetched*. That is, the prefetch requests for these lines are issued one after another without waiting for the earlier ones to complete. Let T_1 denote the full latency of a cache miss and T_{next} denote the latency of an additional pipelined cache miss. Then $T_1 + (w-1) \cdot T_{\text{next}}$ is the cost for servicing all the cache misses for a node with w cache lines. Because on modern processors, T_{next} is much less than T_1, this cost is only modestly larger than the cost for fetching one cache line. On the other hand, having multiple cache lines per node increases its fan-out, and hence can reduce the height of the in-page tree, resulting in better overall performance, as detailed in [6].

Disk-first fpB$^+$-Trees have two kinds of in-page nodes: leaf nodes and nonleaf nodes. Their roles in the overall tree (the disk-optimized view) are very different. While in-page nonleaf nodes contain pointers to other in-page nodes *within the same page*, in-page leaf nodes contain pointers to pages *external to their in-page tree*. Thus, for in-page nonleaf nodes, we pack more entries into each node by using short in-page offsets instead of full pointers. Because all in-page nodes are aligned on cache line boundaries, the offsets can be implemented as a node's starting cache line number in the page. For example, if the cache line is 64 bytes, then a 2 byte offset can support page sizes up to 4MB. On the other hand, in-page leaf nodes contain child page IDs if the page is not a leaf in the overall tree, and tuple IDs if the page is a leaf.

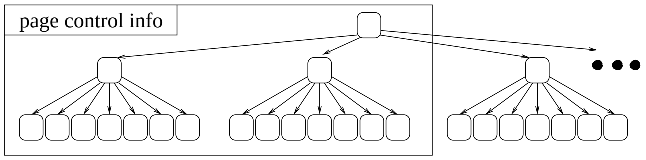
The node size mismatch problem Considering cache performance only, there is an optimal in-page node size, determined by memory system parameters and key and pointer sizes [6]. Ideally, in-page trees



(a) a two-level tree that overflows

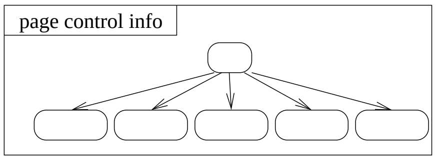


(b) a two-level tree that underflows

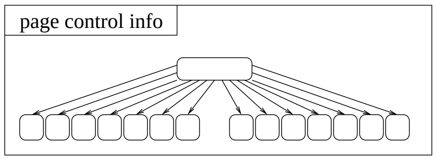


(c) adding a third level to the tree in (b) causes an overflow

Figure 6: The node size mismatch problem



(a) use smaller nodes when overflow



(b) use larger nodes when underflow

Figure 7: Fitting cache-optimized trees in a page

based on this optimal size fit tightly within a page. However, the optimal page size is determined by I/O parameters and disk and memory prices [12, 15]. Thus there is likely a mismatch between the two sizes, as depicted in Figure 6. Figure 6(a) shows an overflow scenario in which a two-level tree with cache-optimal node sizes fails to fit within the page. Figure 6(b) shows an underflow scenario in which a two-level tree with cache-optimal node sizes only occupies half a page, but a three-level tree, as depicted in Figure 6(c), overflows the page. Thus, in most cases, we must give up on having trees with cache-optimal node sizes, in order to fit within the page. (Section 3.2 describes an alternative "cache-first" approach that instead gives up on having the cache-optimized trees fit nicely within page boundaries.)

3.1.1 Determining Optimal In-page Node Sizes

Our goals are to optimize search performance and to maximize page fan-out to preserve good I/O performance. To solve the node size mismatch problem, we give up using cache-optimal node sizes in disk-first fpB$^+$-Trees. In addition, we propose to allow different node sizes for different levels of the in-page tree. As shown in Figure 7, to combat overflow, we can reduce the root node (or restrict its fan-out) as in Figure 7(a). Similarly, to combat underflow, we can extend the root node so that it can have more children, as in Figure 7(b).

But allowing arbitrarily many sizes in the same tree will make index operations too complicated. To keep operations manageable, noting that we already have to deal with different non-leaf and leaf node structures, we instead develop an approach that permits an in-page tree to have two node sizes: one for its leaves and one for its nonleaves. As we shall see, this flexibility is sufficient to achieve our goals.

Optimal node sizes At a high-level, there are three variables that we can adjust to achieve the goals: the number of levels in the in-page tree (denoted L), the number of cache lines of the nonleaf nodes (denoted w) and the number of cache lines of the leaf nodes (denoted x). Here we determine the optimal node sizes for an in-page tree, given the hardware parameters and the page size. Assume we know T_1 is the full latency of a cache miss, and T_{next} is the latency of an additional pipelined (prefetched) cache miss. Then the cost of searching through an L level in-page tree is

$$cost \quad = \quad (L-1)[T_1 + (w-1)T_{\text{next}}] + T_1 + (x-1)T_{\text{next}}$$

We want to select L, w, and x so as to minimize *cost* while maximizing page fan-out.

However, these two goals are conflicting. Moreover, we observed experimentally that because of fixed costs such as instruction overhead, small variations in *cost* resulted in similar search performance. Thus, we combine the two optimization goals into one goal $\mathcal{G}$: maximize the page fan-out while maintaining the analytical search cost to be within 10% of the optimal.

Now we simply enumerate all the reasonable combinations of w and x (e.g., 1-32 lines, thus $32^2 = 1024$ combinations). For each combination, we compute the maximum L that utilizes the most space in the page, which in turn allows *cost* and fan-out to be computed. Table 2 in Section 4 depicts the optimal node widths used in our experiments. Note that the optimal decision is made only once when creating an index. So the cost of enumeration is small.

3.1.2 Operations

Bulkload Bulkloading a tree now has operations at two granularities. At a page granularity, we follow the common B^+-Tree bulkload algorithm with the maximum fan-out computed by our previous computations. Inside each page, we bulkload an in-page tree using a similar bulkload algorithm. For in-page trees of leaf pages, we try to distribute entries across all in-page leaf nodes so that insertions are more likely to find empty slots. But for nonleaf pages, we simply pack entries into one in-page leaf node after another. We maintain a linked list of all in-page leaf nodes of leaf pages in the tree, in order. Note that the depth of an in-page tree, which depends on the number of entries in the page and the bulkload factor, may be less than the maximum depth L computed above. For example, sometimes all the entries in a root page fit into a single node. Then we will build a one-level in-page tree.

Search Two granularities, but straightforward.

Insertion Insertion is also composed of operations at two granularities. If there are empty slots in the in-page leaf node, we insert the entry into the sorted array for the node, by copying the array entries with larger key values to make room for the new entry. Otherwise, we need to split the leaf node into two. We first try to allocate new nodes in the page. If there is no space for splitting up the in-page tree, but the total number of entries in the page is still far fewer than the page maximum fan-out, we reorganize the in-page tree and insert the entry to avoid expensive page splits. But if the total number of entries is quite close to the maximum fan-out (fewer than an empty slot per in-page leaf node), we split the page by copying half of the in-page leaf nodes to a new page and then rebuilding the two in-page trees in their respective pages.

Deletion Deletion is simply a search followed by a lazy deletion of an entry in a leaf node, in which we copy the array entries with larger key values to keep the array contiguous, but we do not merge leaf nodes that become half empty.

3.2 Cache-First fpB^+-Trees

Cache-first fpB^+-Trees start with a cache-optimized B^+-Tree, ignoring page boundaries, and then try to intelligently place the cache-optimized nodes into disk pages. This scheme addresses the node size mismatch problem by giving up putting a whole cache-optimzed tree nicely within every page. The tree node has the common structure of a cache-optimized B^+-Tree node: a leaf node contains an array of keys and tuple IDs, while a nonleaf node contains an array of keys and pointers. However, the pointers in nonleaf nodes are different. Since the nodes are to be put into disk pages, a pointer is a combination of a page ID and an offset in the page, which allows us to follow the page ID to retrieve a disk page and then visit a node in the page by its offset. Nodes are aligned on cache line boundaries, so the in-page offset is the node's starting cache line number in the page.

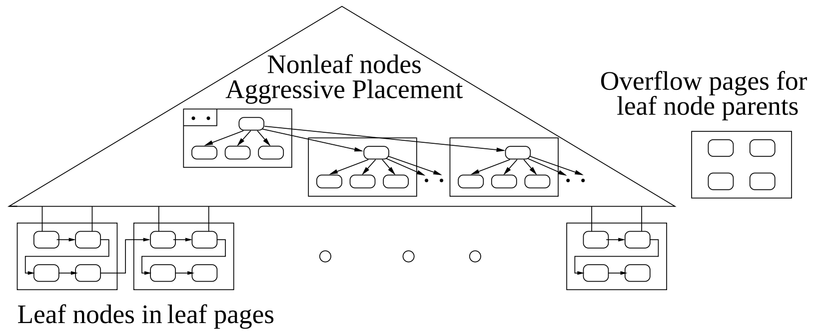


Figure 8: Cache-first fpB$^+$-Tree design

We begin by describing how to place nodes into disk pages in a way that will minimize the structure's impact on disk I/O performance, before presenting our bulkload, insertion, search, and deletion algorithms.

3.2.1 Node Placement

There are two goals in node placement: (1) group sibling leaf nodes together into the same page so that range scans incur fewer disk operations, and (2) group a parent node and its children together into the same page so that searches only need one disk operation for a parent and its child.

To satisfy the first goal, we designate certain pages as leaf pages, which contain only leaf nodes. The leaf nodes in the same leaf page are siblings of one another. This ensures good range scan I/O performance.

Clearly, the second goal cannot be satisfied for all nodes, because only a limited number of nodes fit within a page. Moreover, the node size mismatch problem (recall Figure 6) means that placing a parent and its children in a page almost always results in either an overflow or an underflow for that page. We can often transform a large underflow into an overflow by placing the grandchildren, the great grandchildren, and so on in the same page, until we incur an overflow (see Figures 6(b) and (c)).

There are two approaches for dealing with the overflow. First, an overflowed child can be placed into its own page to become the top-level node in that page. We then seek to place its children in the same page. This aggressive placement helps minimize disk accesses on searches. Second, an overflowed child can be stored in special overflow pages. This is the only reasonable solution for overflowed leaf parent nodes, because their children are stored in leaf pages.

Our node placement scheme is summarized in Figure 8. Leaf nodes are stored in leaf-only pages, for good range scan performance. For nonleaf nodes, we use the aggressive node placement for good search performance, except for leaf parents, which use overflow pages. Moreover, to simplify implementations of aggressive placement, we never place two disjoint subtrees within the same page (which does not improve search I/O performance).

3.2.2 Determining Optimal Node Width

When creating the index, we determine the optimal node widths for cache performance by applying the same optimization goal $\mathcal{G}$ used in the disk-first approach. Assume the node width is w cache lines, each nonleaf cache line can contain m index entries, the full latency and the pipelined latency of a cache miss are T_1 and T_{next}, respectively. Then we compute the search cost as follows (using the similar reasoning to B$^+$-Tree I/O cost computation in [12]):

$$\textit{cost per binary level} \quad = \quad \frac{T_1 + T_{next}(w-1)}{\log_2(m \cdot w)}$$

The numerator is the cost of accessing a nonleaf node. The denominator computes the binary level in a nonleaf node. And their ratio is the search cost per binary level.[5]

[5]This cost computation is a little different from that in [6]. We avoid averaging over all possible numbers of entries in the tree by considering the binary level. This simplifies the computation.

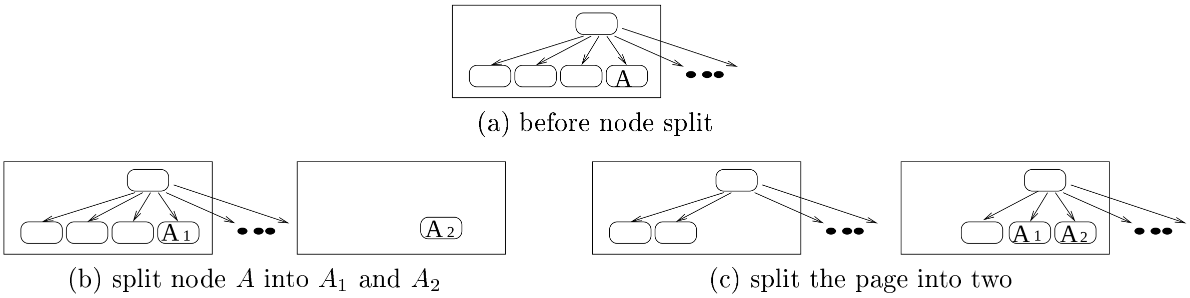


Figure 9: Nonleaf node splits

We enumerate all the reasonable w (e.g. 1-32). For each w, we compute the cost and the number of entries in a leaf page. Then we apply the optimal criterion. Table 2 in Section 4 depicts the optimal node widths used in our experiments.

3.2.3 Algorithms

We now consider each of the index operations.

Bulkload We focus on how to achieve the node placement depicted in Figure 8. Leaf nodes are simply placed consecutively in leaf pages, and linked together with sibling links, as shown in the figure. Nonleaf nodes are placed according to the aggressive placement scheme, as follows.

First, we compute (i) the maximum number of levels of a full subtree that fit within a page, and (ii) the resulting underflow for such a subtree, i.e., how many additional nodes fit within the page. For example, if each node in the full subtree has 69 children, but a page can hold only 23 nodes, then only one level fits completely and the resulting underflow is 22 nodes. We create a bitmap with one bit for each child (69 bits in our example), and set a bit for each child that is to be placed with the parent (22 bits in our example, if we are bulkloading 100% full). We spread these set bits as evenly as possible within the bitmap.

As we bulkload nodes into a page, we keep track of each node's relative level in the page, denoted its *in-page level*. The in-page level is stored in the node header. The top level node in the page has in-page level 0. To place a nonleaf node, we increment its parent's in-page level. If the resulting level is less than the maximum number of in-page levels, the nonleaf node is placed in the same page as its parent, as it is part of the full subtree. If it equals the number, it is placed in the same page if the corresponding bit in the bitmask is set. If it is not set, the nonleaf node is allocated as the top level node in a new page, unless the node is a leaf parent node, in which case it is placed into an overflow page.

Insertion For insertion, if there are empty slots in the leaf node, the new entry is simply inserted. Otherwise, the leaf node needs to be split into two. If the leaf page still has spare node space, the new leaf node is allocated within the same page. Otherwise, we split the leaf page by moving the second half of the leaf nodes to a new page and updating the corresponding child pointers in their parents. (To do this, we maintain in every leaf page a back pointer to the parent node of the first leaf node in the page, and we connect all leaf parent nodes through sibling links.) Having performed the page granularity split, we now perform the cache granularity split, by splitting the leaf node within its page.

After a leaf node split, we need to insert an entry into its parent node. If the parent is full, it must first be split. For leaf parent nodes, the new node may be allocated from overflow pages. But if further splits up the tree are necessary, each new node must be allocated according to our aggressive placement scheme.

Figure 9 helps illustrate the challenges. We need to split node A, a nonleaf node whose children are nonleaf nodes, into two nodes A_1 and A_2, but there is no space in A's page for the additional node. As shown in Figure 9(b), a naive approach is to allocate a new page for A_2. However, A_2's children are half of A's children, which are all top level nodes in other pages. Thus either A_2 is the only node in the new page, which is bad for I/O performance and space utilization, or we must move A_2's children up into A_2's page,

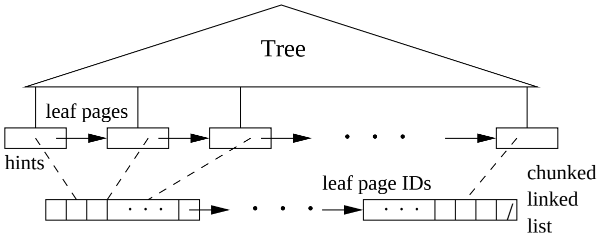


Figure 10: External jump-pointer array

which necessitates promoting A_2's grandchildren to top level nodes on their own pages, and so on. Instead, to avoid the drawbacks of both these options, we split A's page into two, as shown in Figure 9(c). Since we deliberately distributed the co-locating nodes using the bitmask in our bulkload algorithm, a page split is likely to evenly distribute nodes into the two resulting pages.

Finally, for root node splits, we will allocate a new root page and place the newly generated root nodes into this new page until it is full. In the rare case when the new root page is full and the root node splits again, we will rebuild the nonleaf part of the tree. This is because the tree must have grown hundreds of times larger than its orginal size after bulkload and the large number of insertions could make the node placement in the tree suboptimal.

Search Search is quite straightforward. One detail is worth noting. Each time the search proceeds from a parent to one of its children, we compare the page ID of the child pointer with that of the parent page. If the child is in the same page, we can directly access the node in the page without retrieving the page from the buffer manager.

Deletion Similar to disk-first fpB$^+$-Trees.

3.3 Improving Range Scan Performance

For range scans, we employ jump-pointer array prefetching, as described in Section 2.2, for both I/O and cache performance. We now highlight some of the details.

In disk-first fpB$^+$-Trees, both leaf pages and leaf parent pages have in-page trees. For I/O prefetching, we build an internal jump-pointer array by adding sibling links between all in-page leaf nodes that are in leaf parent pages, because collectively these nodes point to all the leaf pages. For cache prefetching, we build a second internal jump-pointer array by adding sibling links between all in-page leaf parent nodes that are in leaf pages, because collectively these nodes point to all the leaf nodes of the overall tree (i.e., all in-page nodes containing tuple IDs). In both jump-pointer arrays, sibling links within a page are implemented as page offsets and stored in the nodes, while sibling links across page boundaries are implemented as page IDs and stored in the page headers.

In cache-first fpB$^+$-Trees, leaf pages contain only leaf nodes, while leaf parent pages can be either in the aggressive placement area or in overflow pages. Thus at both the page and cache granularities, sibling links between leaf parents may frequently cross page boundaries (e.g., a sequence of consecutive leaf parents may be in distinct overlap pages). Thus the internal jump-pointer array approach is not well suited for cache-first fpB$^+$-Trees. Instead, as shown in Figure 10, we maintain an *external* jump-pointer array [6] that contains the page IDs for all the leaf pages, in order to perform I/O prefetching. Similarly, for cache prefetching, we could maintain in each leaf page header an external jump-pointer array, which contains the addresses of all nodes within the page. Instead, we observe that our in-page space managment structure indicates which slots within a page contain nodes, and hence we can use it to prefetch all the leaf nodes in a page before doing a range scan inside the page.

Table 1: Simulation Parameters

Pipeline Parameters	
Clock Rate	1 GHz
Issue Width	4 insts/cycle
Functional Units	2 Integer, 2 FP, 2 Memory, 1 Branch
Reorder Buffer Size	64 insts
Integer Multiply/Divide	12/76 cycles
All Other Integer	1 cycle
FP Divide/Square Root	15/20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	gshare [17]

Memory Parameters	
Line Size	64 bytes
Primary Data Cache	64 KB, 2-way set-assoc.
Primary Instruction Cache	64 KB, 2-way set-assoc.
Miss Handlers	32 for data, 2 for inst.
Unified Secondary Cache	2 MB, direct-mapped
Primary-to-Secondary Miss Latency	15 cycles (plus any delays due to contention)
Primary-to-Memory Miss Latency	150 cycles (plus any delays due to contention)
Main Memory Bandwidth	1 access per 10 cycles

Table 2: Optimal Width Selections (4B keys, $T_1 = 150$, $T_{next} = 10$)

Disk-first fpB+-Trees				
Page Size	Nonleaf Node	Leaf Node	Page Fan-out	cost / optimal
4KB	64B	384B	470	1.06
8KB	192B	256B	961	1.00
16KB	192B	512B	1953	1.03
32KB	256B	832B	4017	1.07

Cache-first fpB+-Trees			
Page Size	Node Size	Page Fan-out	cost / optimal
4KB	576B	497	1.03
8KB	576B	994	1.03
16KB	704B	2001	1.07
32KB	640B	4029	1.05

Micro-indexing			
Page Size	Subarray Size	Page Fan-out	cost / optimal
4KB	128B	496	1.06
8KB	192B	1008	1.06
16KB	320B	2032	1.08
32KB	320B	4064	1.05

4 Experimental Results

In this section, we evaluate the cache and I/O performance of fpB$^+$-Trees. We begin by describing the experimental framework. Then we present our cache performance simulation results and our I/O performance study.

4.1 Experimental Framework

Methodology for Studying Cache Performance We evaluate the CPU cache performance of fpB$^+$-Trees through detailed simulations of fully-functional executables running on a state-of-the-art machine. The simulator models a dynamically-scheduled, superscalar processor similar to the MIPS R10000 [20] running at a clock rate of 1 GHz. The memory hierarchy is based on the Compaq ES40 [9]. We implemented a buffer manager and various index structures (details are below), and ran these on the simulator. The simulator handles I/O reads and writes by making system calls to the underlying operating system. Only user mode executions are simulated. Important simulator parameters are shown in Table 1.[6]

Methodology for Studying I/O Performance We evaluate the I/O performance through experiments on real machines. To study the I/O performance of searches, we executed random searches, and then counted the number of I/O accesses (i.e., the number of buffer pool misses). For searches, the I/O time is dominated by the number of I/Os, because there is little overlap in accessing the pages in a search. To study the I/O performance of range scans, we executed random range scans on an SGI Origin 200 workstation with multiple disks. Furthermore, we evaluate the I/O performance of range scans in a commerical DBMS: we implemented our jump-pointer array scheme within DB2, and executed range scan queries on DB2. Details on our Origin and DB2 experiments are provided later in the subsections describing the range scan I/O performance results.

Implementation Details Our buffer manager uses the CLOCK algorithm to do page replacement. On top of this buffer manager, we implemented four index structures: i) disk-optimized B$^+$-Trees, ii) micro-indexing, iii) disk-first fpB$^+$-Trees, and iv) cache-first fpB$^+$-Trees. We wrote bulkload, search, insertion,

[6]The simulation model and parameters match those in [6].

deletion, and range scan implementations for all the trees (range scans for micro-indexing was not explicitly implemented because its behavior is similar to that of disk-optimized B^+-Trees).

We use 4 byte page IDs, 4 byte tuple IDs, and 2 byte in-page offsets. We performed experiments with 4 byte keys and 20 byte keys. We will present the 4 byte key experiments first and discuss our 20 byte key experiments in Section 4.4. We partitioned keys and pointers into separate arrays in all tree nodes for better cache performance [11, 16]. Disk-first fpB^+-Trees have 2 byte in-page pointers in nonleaf nodes and 4 byte pointers in leaf nodes, while cache-first fpB^+-Trees have 6 byte pointers combining page IDs and in-page offsets in nonleaf nodes. We performed experiments for page sizes of 4KB, 8KB, 16KB, and 32KB, which covers the range of page sizes in most of today's database systems. As shown in Table 2, we computed the optimal node widths for fpB^+-Trees using $T_1 = 150$ and $T_{next} = 10$ from Table 1 and when key size is 4 bytes.

In our micro-indexing implementation, a tree page contains a header, a micro-index, a key array, and a pointer array. The micro-index is formed by dividing the key array into sub-arrays of the same size and copying their first keys. A search in a page first looks up the micro-index to decide which sub-array to go to and then searches that sub-array. For better performance, we require the sub-array size to be a multiple of the cache line size (if applicable) and align the key array at cache line boundaries. To improve the performance of micro-indexing, we employ pB^+-Tree-like prefetching for micro-indexes, key sub-arrays, and pointer sub-arrays. Insertion and deletion follow the algorithms of disk-optimized B^+-Trees, but then rebuild the affected parts of the micro-index. As shown in Table 2, we computed the optimal sub-array sizes for micro-indexing based on the same optimal criterion as advocated for fpB^+-Trees: maximize page fan-out while keeping the analytical search cost to within 10% of the optimal.

We try to avoid conflict cache misses in the buffer manager between buffer control structures and buffer pool pages. The control structures are allocated from the buffer pool itself, and only those buffer pages that do not conflict with the control structures will be used. In fpB^+-Trees, putting top-level in-page nodes at the same in-page position would cause cache conflicts among them. So we instead place them at different locations determined by a function of the page IDs.

4.2 Cache Performance

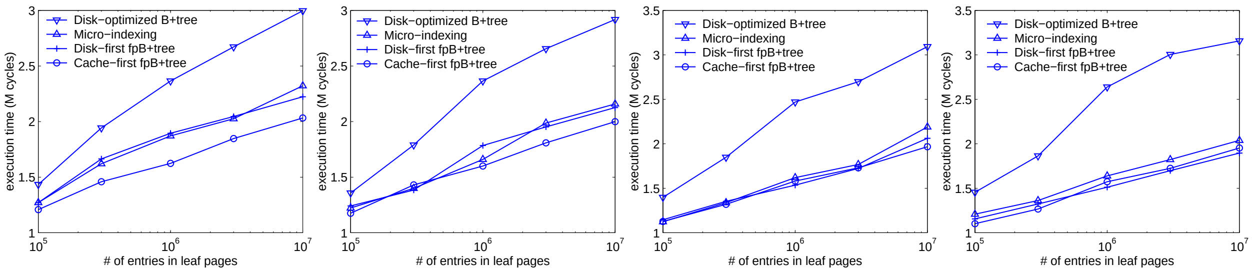
4.2.1 Search Performance

Varying the number of entries in leaf pages Figures 11 and 12 show the execution times of 2000 random searches after bulkloading 100K, 300K, 1M, 3M, and 10M keys into the trees (nodes are 100% full except the root). All caches are cleared before the first search, and then the searches are performed one immediately after another. The four plots in Figure 11 show search performance when the database page sizes are 4KB, 8KB, 16KB, and 32KB, respectively. The fpB^+-Trees and micro-indexing use the corresponding optimal widths in Table 2. From the figures, we see that the cache-sensitive schemes, fpB^+-Trees and micro-indexing, all perform significantly better than disk-optimized B^+-Trees, achieving speed-ups between 1.09 and 1.77 at all points and between 1.25 and 1.77 when the trees contain at least 1M entries. Moreover, comparing the three cache-sensitive schemes, we find their performance more or less similar. When the page size is 4KB, the cache-first fpB^+-Tree is slightly better than the other two. But for the other page sizes, their performance is very close.

When the page size increases from 4KB to 32KB, the performance of disk-optimized B^+-Trees becomes slightly worse. While larger leaf pages cause more cache misses at the leaf level, this cost is partially compensated by the savings at the nonleaf levels: trees become shallower and/or root nodes have fewer entries. At the same time, fpB^+-Trees and micro-indexing perform better because larger page sizes leave more room for optimization. With the two trends, we see larger speed-ups: over 1.41 for 16KB pages, and over 1.54 for 32KB pages, when trees contain at least 1M entries.

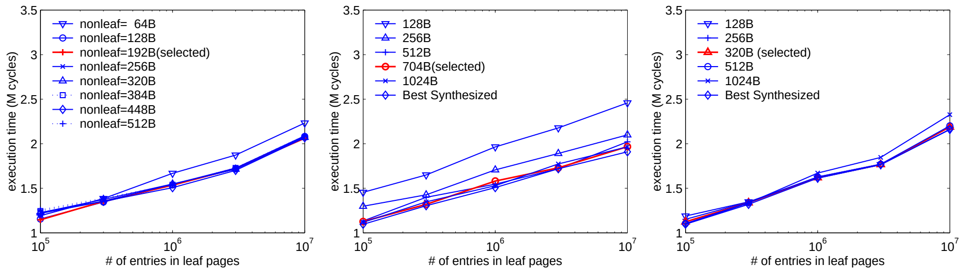
Figure 12 compares the performance of different widths for fpB^+-Trees and micro-indexing when the page size is 16KB. Recall that our optimal criterion is to maximize leaf page fan-out while keeping analytical search performance within 10% of the best. Figure 12 confirms that our selected trees indeed achieve search performance very close to the best among the node choices.

Figure 12(a) shows the performance of disk-first fpB^+-Trees using nonleaf node sizes from 64B (an L2 cache line) to 512B (8 L2 cache lines). Our selected optimal tree is within 2% of the best execution times.



(a) page size = 4KB (b) page size = 8KB (c) page size = 16KB (d) page size = 32KB

Figure 11: 2K searches after bulkloading 100K-10M keys 100% full.



(a) Disk-first fpB$^+$-Tree (b) Cache-first fpB$^+$-Tree (c) Micro-indexing

Figure 12: Optimal width selection (16KB page, 4B key)

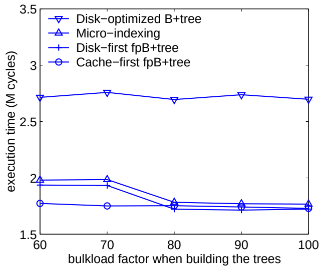
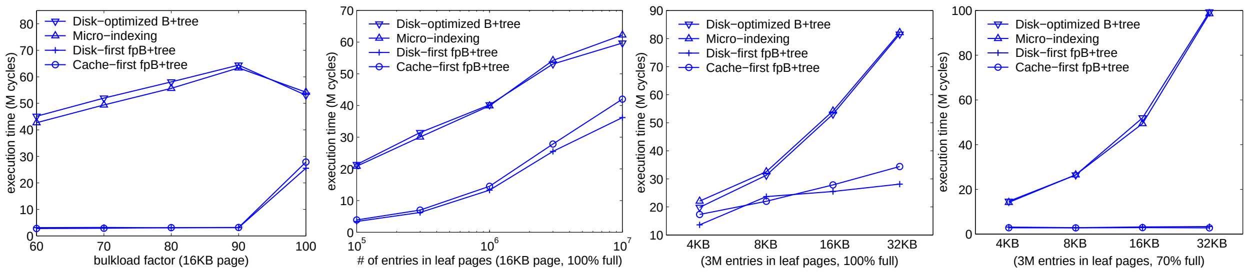


Figure 13: Search performance varying bulkload factors (16KB page, 3M entries in leaf pages)

For cache-first fpB$^+$-Trees we measured the performance for node sizes ranging from 128B to 1024B. In Figure 12(b), for simplicity, we only show curves for node sizes of 128B, 256B, 512B, 704B, 1024B, and a best performance curve synthesized by taking the minimums of all curves with the same # of entries in leaf pages. Our selected optimal tree performs within 5% of the best. Figure 12(c) shows the performance curves of micro-indexing with different sub-array sizes. As in the comparisons of Figure 12(b), we measured performance for sub-array sizes from 128B to 1024B, but for simplicity, the figure only shows curves for five different sizes, and the best synthesized curve. Our selected optimal structure performs within 3% of the best performance. In the experiments of Section 4.2 and 4.3, we use the optimal sizes given in Table 2.

Varying the bulkload factor In Figure 13, we varied the 3M-entry experiments in Figure 11(c) with bulkload factors ranging from 60% to 100%. Compared with disk-optimized B$^+$-Trees, fpB$^+$-Trees and micro-indexing achieve speed-ups between 1.37 and 1.60.

The step-down at 80% for micro-indexing and disk-first fpB$^+$-Trees is because they reduce one page level



(a) Varying bulkload factors (b) Varying # entries (c) Varying page sizes (100% full) (d) Varying page sizes (70% full)

Figure 14: Insertion Performance

at 80%. Although the disk-optimized B^+-Trees also reduce one level here, the savings are offset by a larger cost for searching leaf pages with increased bulkload factors. Cache-first fpB^+-Trees all have the same number of node levels in this set of experiments and therefore similar performance.

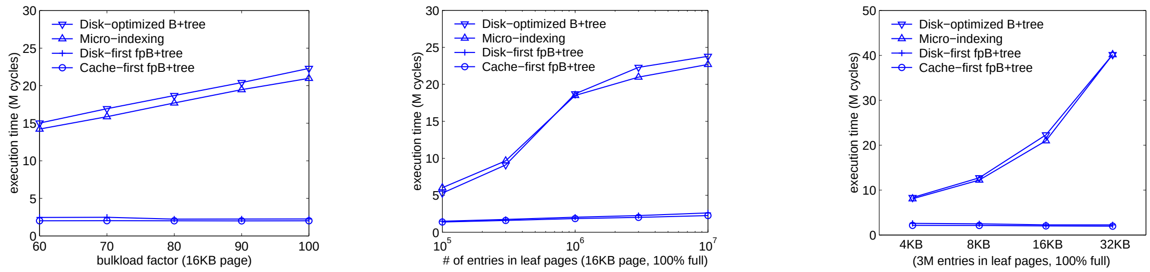
4.2.2 Insertion Performance

Figure 14 shows the insertion performance in four different settings. The experiments all measured the execution times for inserting 2000 random keys after bulkloads, while varying the bulkload factor, the numbers of entries in leaf pages, and the page size. The fpB^+-Trees achieve up to a 35-fold speed-up over disk-optimized B^+-Trees, while micro-indexing performs almost as poorly as disk-optimzed B^+-Trees.

Figure 14(a) compares insertion performance of trees from 60% to 100% full containing 3M keys. Compared to disk-optimized B^+-Trees, the fpB^+-Trees achieve 14 to 20-fold speed-ups between 60% and 90%, while for 100% full trees, they are over 1.9 times better. Interestingly, the curves have extremely different shapes: those of disk-optimized B^+-Trees and micro-indexing increase from 60% to 90% but drop at the 100% point, while the curves of fpB^+-Trees stay flat at first but jump dramatically at the 100% point. These effects can be explained by the combination of two factors: data movement and page splits. When trees are 60% to 90% full, insertions usually find empty slots and the major operation after searching where the key belongs is to move the key and pointer arrays in order to insert the new entry. In disk-optimized B^+-Trees, this data movement is by far the dominant cost. As the occupied portions of the arrays grow from 60% to 90%, this cost increases, resulting in larger insertion times. Micro-indexing keeps the same large array structure untouched and therefore suffers from the same effect. However, in fpB^+-Trees, we reduced the data movement cost by using smaller cache-optimized nodes, resulting in 14 to 20-fold speed-ups. Data movement has become much less costly than search, leading to the flat curves up through 90% full. When the trees are 100% full, insertions cause frequent page splits. In fpB^+-Trees, the cost of a page split is far more than the previous data movement cost, resulting in the large jump seen in the curves. In B^+-Trees and micro-indexing, however, the page split cost is comparable to copying half of a page, which is the average data movement cost for inserting into an almost full page. But later insertions may hit half empty pages (just split) and hence incur less data movement, resulting in faster insertion times at the 100% point.

Figure 14(b) shows insertion performance on full trees of different sizes. Compared to disk-optimized B^+-Trees, fpB^+-Trees achieve speed-ups from 6.26 to 1.42 when the number of entries in leaf pages is increased from 100K to 10M. This decrease in speed-up is caused by the increasing number of page splits (from 48 to 1631 leaf page splits for disk-optimized B^+-Trees, and similar trends for other indexes). As argued above, increased page splits have a much greater performance impact on fpB^+-Trees than on disk-optimized B^+-Trees and micro-indexing, leading to the speed-up decrease.

Figures 14(c) and (d) compare the insertion performance varying page sizes when trees are 100% and 70% full. As the page size grows, the execution times of disk-optimized B^+-Trees and micro-indexing explode because of the combined effects of larger data movement and larger page split costs. In fpB^+-Trees, though page split costs also increase, search and data movement costs only change slightly, because with larger page sizes comes the advantages of larger optimal node widths. Therefore the curves of fpB^+-Trees increase in



(a) Varying bulkload factors (b) Varying # of entries (100% full) (c) Varying page sizes (100% full)

Figure 15: Deletion Performance

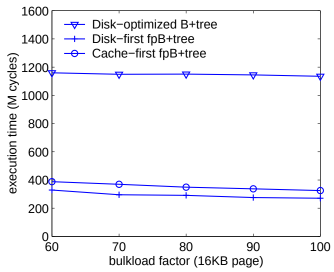


Figure 16: Range Scan Performance (16KB page, scanning 1M keys)

Figure 14(c) but are almost flat in (d). Altogether in Figure 14(c) and (d), fpB$^+$-Trees achieve 1.15–2.90 and 4.67–35.6 fold speed-ups over disk-optimized B$^+$-Trees, respectively.

Comparing the two fpB$^+$-Trees, we see they have similar insertion performance. Sometimes cache-first fpB$^+$-Trees perform worse than disk-first fpB$^+$-Trees. This is primarily because of the more complicated node/page split operations in cache-first fpB$^+$-Trees, as discussed in Section 3.2.

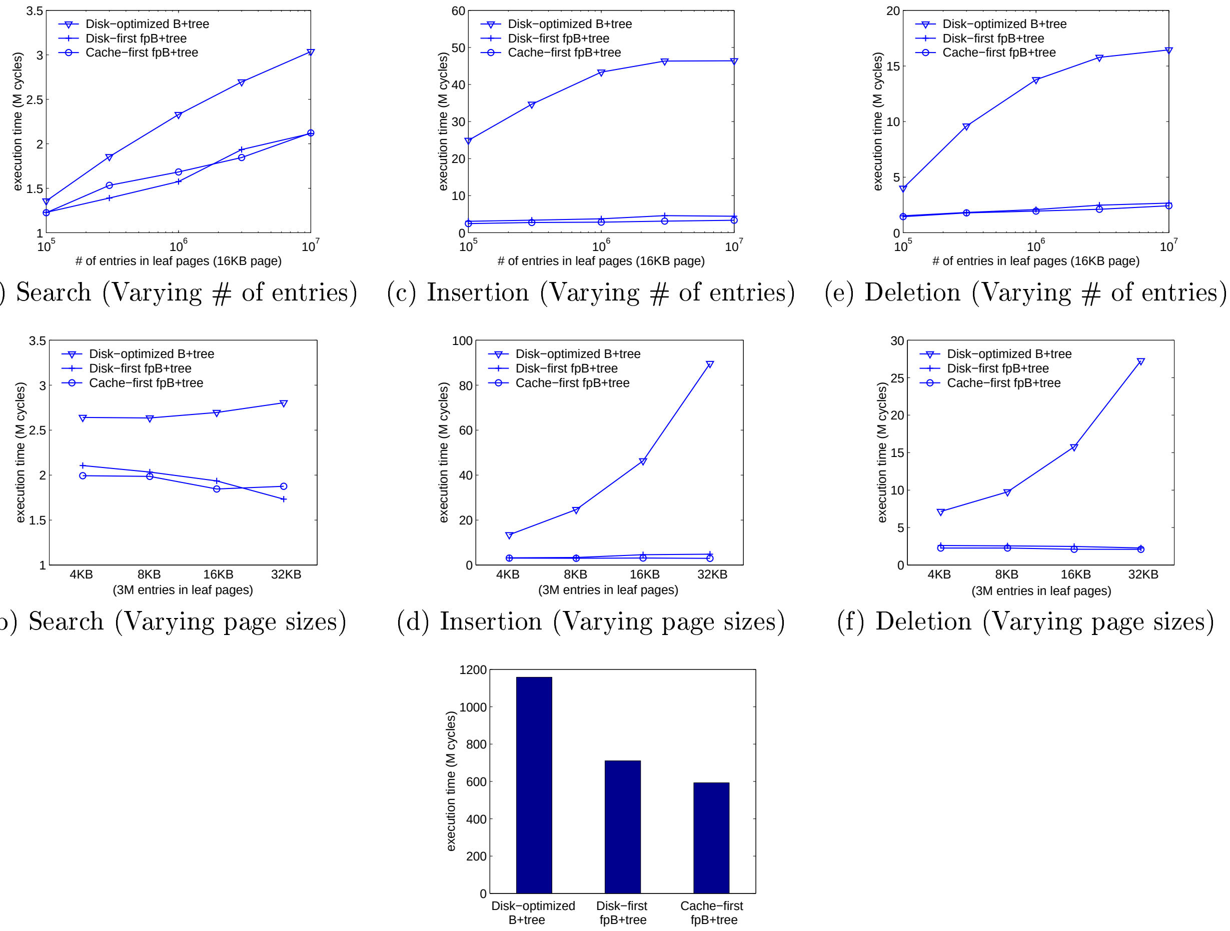
4.2.3 Deletion Performance

Deletions are implemented as lazy deletions in all the indexes. A search is followed by a data movement operation to remove the deleted entry, but we do not merge underflowed pages or nodes. Figure 15 evaluates deletion performance (for 2000 random deletions) in three settings: (a) varying the bulkload factor when the page size is 16KB, (b) varying the number of entries bulkloaded, and (c) varying the page sizes when the trees are 100% full. The dominant cost in disk-optimized B$^+$-Trees and micro-indexing is the data movement cost, which increases as the bulkload factor increases and the page size grows. However, the search and data movement costs of fpB$^+$-Trees only change slightly. So the fpB$^+$-Trees achieve 3.2–20.4 fold speed-ups over disk-optimized B$^+$-Trees.

4.2.4 Range Scan Performance

Figure 16 compares the range scan cache performance of fpB$^+$-Trees and disk-optimized B$^+$-Trees. The trees are bulkloaded with 3M keys varying bulkload factors from 60% to 100%. We generate 100 random start keys, for each computing an end key such that the range spans precisely 1M tuple IDs, and then perform these 100 range scans one after another. Compared to the disk-optimized B$^+$-Trees, the disk-first and cache-first fpB$^+$-Trees achieve speed-ups 3.5-4.2 and 3.0-3.5, respectively[7].

[7] Our paper [7] reported up to a factor 2.3 speedup for the 100% experiments. We found a performance bug after looking into the statistics collected by the simulator. We did not issue cache prefetch instructions for leaf node header fields, which are visited by the range scan to obtain the number of index entries in the nodes. After fixing the bug, we see larger speedups.



(a) Search (Varying # of entries) (c) Insertion (Varying # of entries) (e) Deletion (Varying # of entries)

(b) Search (Varying page sizes) (d) Insertion (Varying page sizes) (f) Deletion (Varying page sizes)

(g) Range Scan (16KB page, scanning 1M keys)

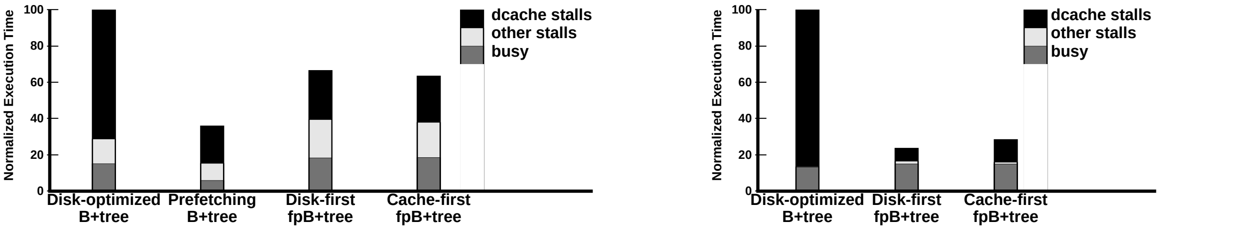
Figure 17: Mature Tree Cache Performance

In disk-optimized B$^+$-Trees, a range scan sequentially visits the large tuple ID array when visiting a leaf page. In contrast, range scans in fpB$^+$-Trees have to deal with node structures within leaf pages, potentially leading to worse cache performance. But using prefetching for range scans, we can still achieve up to 4-fold speedups despite the added complexity.

When the bulkload factor increases from 60% to 100%, the execution times of range scan in disk-optimized B$^+$-Trees are almost the same. This is because the number of memory accesses does not change significantly: more tuple IDs are read in every leaf page, but fewer leaf pages are visited. However, the execution times in fpB$^+$-Trees decrease with the increase of bulkload factor. In our implementation, for simplicity, we always prefetch the whole array of tuple IDs in each leaf node no matter how full it is. So when the bulkload factor increases, fewer leaf nodes are visited and therefore there is less prefetching overhead. This may be improved by maintaining statistics and dynamically determining how many lines in a leaf node to prefetch for each range scan operation.

4.2.5 Mature Tree Cache Performance

We also performed a set of experiments with mature trees, created by bulkloading 10% of index entries and then inserting the remaining 90%. As shown in Figure 17, we find similar performance gains to our previous experiments. Search is improved by a factor of 1.1-1.6, insertion by a factor of 4-30, deletion by a factor of 2.6-13, and range scan by 1.63 and 1.95 (for disk-first and cache-first fpB$^+$-Trees respectively).



(a) Search (16KB page, 2K random searches) (b) Range Scan (16KB page, range contains 1M keys)

Figure 18: Normalized Execution Time Breakdowns

4.2.6 Execution Time Breakdowns

We finish the discussion of cache performance in this subsection with the execution time breakdowns of two representative experiments: one for index search and one for index range scan. Figure 18(a) corresponds to the experiments shown earlier in Figure 11(c) with 10M entries bulk-loaded, and Figure 18(b) corresponds to the experiments shown earlier in Figure 16 when bulkload factor is 100%.

Each bar in Figure 18 represents execution time normalized to a disk-optimized B$^+$-Tree, and is broken down into the following three categories that explain what happened during all potential *graduation slots*.[8] The bottom section (*busy*) is the number of slots where instructions actually graduate. The other two sections are the number of slots where there is no graduating instruction, broken down into data cache stalls and other stalls. Specifically, the top section (*dcache stalls*) is the number of such slots that are immediately caused by the oldest instruction suffering a data cache miss, and the middle section (*other stalls*) is all other slots where instructions do not graduate.

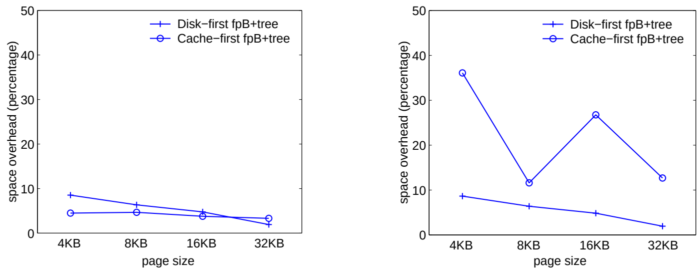
As shown in Figure 18, our fpB$^+$-Trees significantly reduce the amount of exposed miss latency (i.e. the *dcache stalls* component). In the index search experiments, fpB$^+$-Trees eliminate more than 60% of the data cache stall time. And in the index range scan experiments, more than 85% of the data cache stalls have been eliminated. Interestingly the busy times and the other stalls of fpB$^+$-Trees increase slightly. This can be explained by the fact that fpB$^+$-Trees build more structures inside a disk page. So more instructions are needed for index operations, increasing the number of graduated instructions (busy times). Furthermore, more instructions lead to more resource and other contentions, which are reflected in the other stalls. However, these increases are overshadowed by the savings in data cache stalls. Overall, the disk-first and cache-first fpB$^+$-Trees speed up index search by a factor of 1.50 and 1.57, respectively. They speed up index range scan by a factor of 4.2 and 3.5, respectively. These results demonstrate that the fpB$^+$-Trees speedups indeed come from a significant reduction in the exposed miss latency.

4.3 I/O Performance and Space Overhead

Space Overhead Figure 19 shows the space overhead[9] of the fpB$^+$-Trees compared to disk-optimized B$^+$-Trees for a range of page sizes, depicting two (extremal) scenarios: (a) immediately after bulkloading the trees 100% full, and (b) after inserting 9M keys into trees bulkloaded with 1M keys. We see that in each of these scenarios, disk-first fpB$^+$-Trees incur less than a 9% overhead. In cache-first fpB$^+$-Trees, the space overhead is less than 5% under scenario (a), even better than disk-first fpB$^+$-Trees. This is because the leaf pages in cache-first fpB$^+$-Trees only contain in-page leaf nodes, while disk-first fpB$^+$-Trees build in-page trees (containing nonleaf and leaf nodes) in leaf pages. However, for the mature tree scenarios, the space overheads of the cache-first fpB$^+$-Tree can grow to 36%, because of the difficulties in maintaining effective placement of nodes within pages over many insertions.

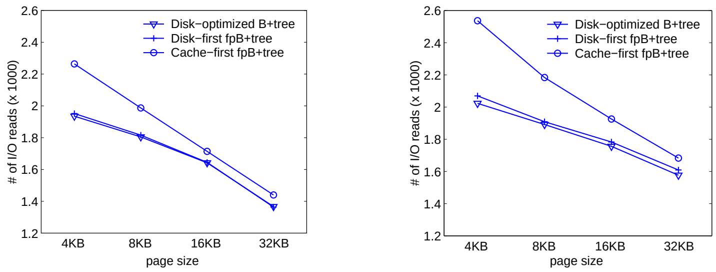
[8]The number of graduation slots is the issue width (4 in our simulated architecture) multiplied by the number of cycles. We focus on graduation slots rather than issue slots to avoid counting speculative operations that are squashed.

[9]Space Overhead $= \frac{\#\text{ of pages in the index}}{\#\text{ of pages in a disk-optimized B}^+\text{-Tree}} - 1$



(a) After bulkloading trees 100% full (b) Mature trees

Figure 19: Space Overhead



(a) search after bulkload (b) search in mature trees

Figure 20: I/O Search performance

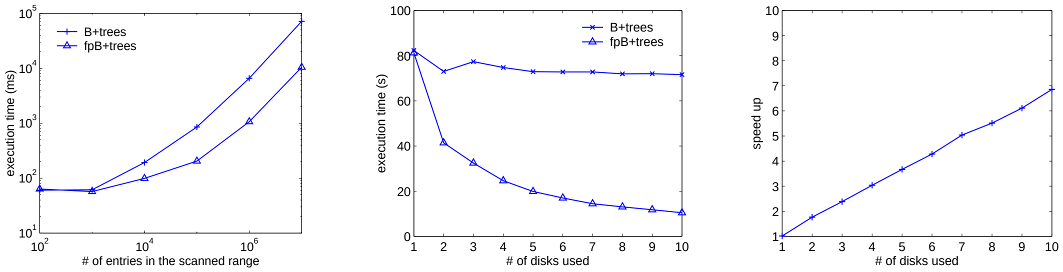
Figure 19 also shows that as the page size grows, the space overhead of disk-first fpB$^+$-Trees decreases because larger pages allow more freedom when optimizing in-page node widths.

4.3.1 Search Performance

Figure 20 shows the search I/O performance of fpB$^+$-Trees. The figure reports the number of I/O page reads that miss the buffer pool when searching 2000 random keys in trees containing 10M keys. The buffer pool was cleared before every experiment. We see that for all page sizes, disk-first fpB$^+$-Trees perform close to that of disk-optimized B$^+$-Trees, accessing less than 3% more pages. However, cache-first fpB$^+$-Trees may access up to 25% more pages. After looking into the experiments, we determined that the extra cost is incurred mainly when accessing leaf parent nodes in overflow pages. For example in the 4KB case in Figure 20(a), the fan-out of a nonleaf node is 57 and a page can contain part of a two-level tree. But only 6 out of the 57 children can reside on the same page as a node itself. Therefore even if all the parents of the leaf parent nodes are top-level nodes, 51 out of every 57 leaf parent nodes will still be placed in overflow pages, leading to many more page reads than disk-optimized B$^+$-Trees. However, as page sizes grow, this problem may be alleviated, as can be seen for the 32KB points.

4.3.2 Range Scan Performance on Real Hardware

Unlike our search experiments, which counted the number of I/O accesses, our range scan I/O performance experiments measure running times on real hardware. Figure 21 shows the I/O performance of fpB$^+$-Trees vs. B$^+$-Trees for range scans, on an SGI Origin workstation running Irix 6.5 with four 180MHz MIPS R10000 processors, 128MB RAM, and 12 SCSI disks. Each disk is a Seagate Cheetah 4LP with a maximum transfer rate of 40 Mbytes/sec and a track-to-track seek type of 18 msec (typical). We imitate raw disk partitions by allocating a large file on each disk and managing the mapping from page IDs to file offsets ourselves. The file system uses 16KB disk blocks, so accordingly, we set the tree page size to be 16KB. Our buffer manager has a



(a) Execution time vs. range size (b) time vs. # of disks (c) speed-up vs. # of disks

Figure 21: Range Scan I/O Performance

(a) varying # of prefetchers (SMP degree = 9) (b) varying SMP degree (# of prefetchers = 8)

Figure 22: Impact of range scan prefetching on the performance of DB2

dedicated thread for each of the disks, which performs I/O operations on behalf of the operation requesters.

For these experiments, we bulkloaded the trees 100% full with 90 million keys and then inserted 10 million keys to make the trees mature. We performed various range scan operations on the mature trees. Each reported data point is the average of 10 trials.

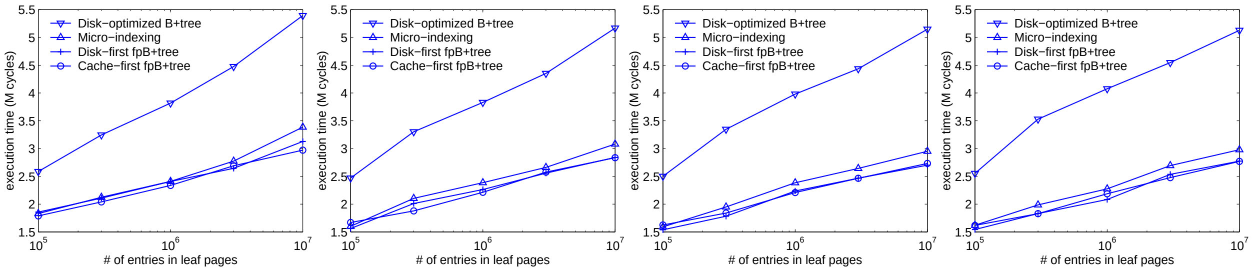
Figure 21(a) shows the execution time in milliseconds for range scans using 10 disks, where the starting keys are selected at random and the size of the range varies from 10^2 to 10^7 entries. Note that a 16KB leaf page can hold more than 2000 entries, or more than 1400 entries when the tree is 70% full, which is typically the case. Thus for small ranges (10^2 and 10^3), the execution times for the two trees are indistinguishable. For larger ranges (10^4 and up), multiple pages are needed, and jump-pointer array prefetching provides a significant improvement over the B^+-Tree. Even for the 10^4 case, which scans only a few pages, the fpB^+-Tree is 1.9 times faster than the B^+-Tree. Better still, for large scans of 10^6–10^7 entries, the fpB^+-Tree is *6.2-6.9 times faster* than the B^+-Tree.

From the small range results in Figure 21(a), we see that our technique for avoiding overshooting is quite effective. When ranges are small, there is almost no additional I/O overhead for searching end keys, since end keys often reside in the same leaf pages as begin keys. Even when they are in different leaf pages, the end leaf page, which has been fetched into the buffer pool by the search, will likely still be in the buffer pool when needed for the scan.

Figure 21(b) shows the execution time in seconds for large range scans (10^7 entries), varying the number of disks. Figure 21(c) shows the corresponding speed-ups. We see the trend of decreasing execution time (and hence increasing speed-up) with increasing numbers of disks. When we increase the number of disks from 1 to 10, we see an almost linear increase in speed-up from 1 to 6.9.

Table 3: Optimal Width Selections (20B keys, $T_1 = 150$, $T_{next} = 10$)

Disk-first fpB$^+$-Trees					Cache-first fpB$^+$-Trees				Micro-indexing			
Page Size	Nonleaf Node	Leaf Node	Page Fan-out	cost optimal	Page Size	Node Size	Page Fan-out	cost optimal	Page Size	Subarray Size	Page Fan-out	cost optimal
4KB	128B	640B	156	1.08	4KB	448B	162	1.00	4KB	320B	160	1.00
8KB	256B	704B	319	1.05	8KB	896B	333	1.07	8KB	704B	332	1.10
16KB	512B	704B	638	1.02	16KB	704B	667	1.03	16KB	896B	668	1.06
32KB	576B	1280B	1325	1.06	32KB	704B	1334	1.03	32KB	1088B	1344	1.08



(a) page size = 4KB (b) page size = 8KB (c) page size = 16KB (d) page size = 32KB

Figure 23: 2K searches after bulkloading 100K-10M 20B-keys 100% full.

4.3.3 Range Scan Performance on a Commercial DBMS

To evaluate the impact of range scan prefetching on a commerical DBMS, we implemented our jump-pointer array scheme within IBM's DB2 Universal Database[10]. Because DB2's index structures support reverse scans and SMP scan parallelism, we added links in both directions, and at all levels of the tree. These links are adjusted at every non-leaf page split and page merge.

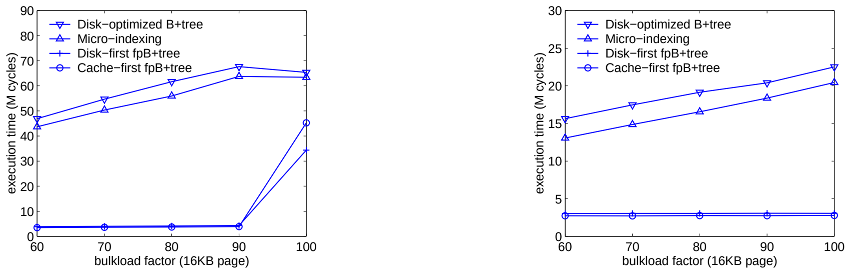
We performed experiments on an IBM 7015-R30 machine (from the RS/6000 line) with 8 processors, 80 SSA disks, and 2GB of memory, running the AIX operating system. We populated a 12.8 GB table across 80 raw partitions (i.e., 160 MB per partition) using 10 concurrent processes to insert a total of roughly 50 million rows of random data of the form `(int,int,char(20),int,char(512))`. An index was created using the three integer columns; its initial size was less than 1 GB, but it grows through page splits. We used the query `SELECT COUNT(*) FROM DATA`, which is answered using the index. Figure 22 shows the results of these experiments.

As we see in Figure 22, our results on an industrial-strength DBMS are surprisingly good (2.5-5.0 speedups). The top curves in both figures are for the plain range scan implementation without jump-pointer array prefetching. The bottom curves show the situation when the leaf pages to be scanned are already in memory; this provides a limit to the possible performance improvements. Figure 22(a) shows that the performance of jump-pointer array prefetching increases with the number of I/O prefetchers, until the maximum performance is nearly reached. Figure 22(b) shows that increasing the degree of parallelism increases the query performance, which again tracks the maximum performance curve.

4.4 Experimental Results with 20B keys

To study the performance of fpB$^+$-Trees with larger key sizes, we performed a set of experiments with 20-byte keys in this section. Table 3 shows the optimal width selections for fpB$^+$-Trees and micro-indexing when key size is 20 bytes. As shown in Figure 23, Figure 24, and Figure 25, we find similar cache performance

[10]Notices, Trademarks, Service Marks and Disclaimers: The information contained in this publication does not include any product warranties, and any statements provided in this document should not be interpreted as such. The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries: IBM, DB2, DB2 Universal Database. Other company, product or service names may be the trademarks or service marks of others.



(a) 2K insertions varying bulkload factors (b) 2K deletions varying bulkload factors

Figure 24: Update performance when key size is 20B

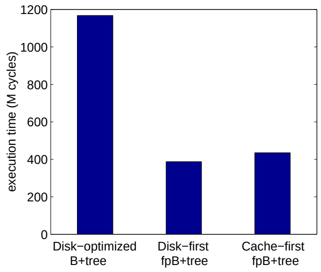


Figure 25: Range Scan Performance (16KB page, scanning 1M keys, key size is 20B)

gains to our previous experiments. All three cache-sensitive index schemes perform significantly better in search than disk-optimized B$^+$-Trees. However, micro-indexing does not solve the data movement problem and therefore still has poor insertion and deletion performance. With fpB$^+$-Trees, search is improved by a factor of 1.39-1.96, insertion by a factor of 1.4-17, deletion by a factor of 5-8, and range scan by a factor of 3.0 and 2.7 for disk-first and cache-first fpB$^+$-Trees, respectively.

Comparing Figure 23 to Figure 11 in Section 4.2.1, we see a large increase in the execution times of disk-optimized B$^+$-Trees. With larger keys, the tree pages have smaller fan-outs and the trees become deeper. Key comparison costs increase. In addition, a cache line contains fewer keys. So the savings due to spatial locality at the end of a binary search decreases. In fpB$^+$-Trees, we use prefetching to bring in all cache lines of a cache-optimized node in parallel. This ability to use nodes much larger than a cache line enables fpB$^+$-Trees to effectively improve cache performance not only for small key sizes but also for large key sizes. Compared to disk optimized B$^+$-Trees, the performance deterioration due to larger key sizes of fpB$^+$-Trees is also less dramatic, thus leading to even larger speedups in search performance with 20 byte keys.

Comparing the update performance with previous figures, we see that the execution times of disk-optimized B$^+$-Trees almost stay the same. This is because larger key sizes do not change the data movement costs and page split cost.

5 Discussion

We have focused on improving performance for B$^+$-Trees with fixed length keys. Although B$^+$-Trees with variable length keys have different page structures, we can still break disk-optimized pages into cache-optimized nodes and employ fpB$^+$-Trees to improve cache and disk performance.

As shown in Figure 26(a), a common way to deal with variable length entities in database systems is to use slotted pages. A page (tree node) contains a header at the beginning, space to allocate index entries in the middle, and an array of slots pointing to the index entries in the key order in the end. The indirection provided by the slot array allows index entries to be allocated in any order. This partially solves the data

(a) Using slotted page (b) Extracting key prefixes into an offset array

Figure 26: B^+-Tree page structures for variable length keys

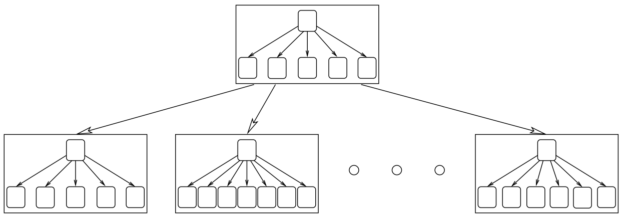


Figure 27: fpB^+-Trees for variable length keys

movement problem in the continuous array structure: index entries do not need to be moved for insertion and deletion, instead we have to move the slot array. However, there will not be any spatial locality across index entries for search now (not even the limited spatial locality as in the continuous array case when a binary search narrows down to within a cache line). So search performance will get even worse. The structure shown in Figure 26(b) has been suggested to have better cache performance in [11, 16]. The main idea is to extract fixed length prefixes of the variable length keys to form a prefix array along with byte offsets pointing to the rest of index entries, which includes the rest of the keys and page IDs or tuple IDs. If the prefixes distinguish keys well, key comparison will mostly use the prefixes without accessing the rest of the keys. And we essentially get back to the continous array structure. Better performance results from the savings in key comparison and the spatial locality of the key prefix array.

fpB^+-Trees can be employed to improve performance of B^+-Trees with variable length keys. Since prefetching allows cache optimized nodes to be much larger than a cache line, we will be able to put large variable length index entries in cache optimized nodes and then embed cache optimized nodes into disk-optimized B^+-Trees in either disk-first or cache-first way, as shown conceptually in Figure 27. The actual fan-out of a node varies due to the variable sizes of its index entries. Alternatively, we can replace the key prefix array in the structure of Figure 26(b) with an in-page cache-optimized tree to further improve search and update performance. It will be interesting to implement both schemes and compare their performance.

For I/O range scan prefetching, we build jump pointer arrays. An alternative approach traverses the nonleaf pages of a tree, retrieving leaf page IDs for prefetching. The path from root page to the current leaf parent page can be recorded temporarily and with the help of the child pointers in nonleaf pages, one can sweep the path across the tree from the beginning to the end of the range. Although this approach saves the effort of building additional data structures, it becomes complicated when there are more than three levels in trees. Furthermore, traversal of the nonleaf part may frequently across page boundaries for cache-first fpB^+-Trees.

6 Conclusions

Previous studies on improving index performance have focused either on optimizing the cache performance of memory-resident databases, or else optimizing the I/O performance of disk-resident databases. What has been lacking prior to this study is an index structure that achieves good performance for both of these important levels of the memory hierarchy. Our experimental results in this paper demonstrate that *Fractal Prefetching B$^+$-Trees* are such a solution. They achieve large gains in cache performance compared with disk-optimized B$^+$-Trees for searches, range scans, and updates on modern systems. Moreover, they provide up to a fivefold improvement in the I/O performance of range scans on a commercial DBMS (DB2). Comparing the two fpB$^+$-Tree approaches, we recommend in general the *disk-first* approach, for its minimal I/O impact. But if there is sufficient memory to hold most of the index pages, we recommend the *cache-first* approach, for its slightly better cache performance. By effectively addressing the complete memory hierarchy, fpB$^+$-Trees are a practical solution for improving DBMS performance.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th VLDB*, pages 169–180, Sept. 2001.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th VLDB*, pages 266–277, Sept. 1999.

[3] L. A. Barroso, K. Gharachorloo, and E. D. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th ISCA*, pages 3–14, June 1998.

[4] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. In *Proceedings of the 41st IEEE FOCS*, pages 399–409, Nov. 2000.

[5] P. Bohannon, P. McIlroy, and R. Rastogi. Improving Main-Memory Index Performance with Partial Key Information. In *Proceedings of the 2001 SIGMOD Conference*, May 2001.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance through Prefetching. In *Proceedings of the SIGMOD 2001 Conference*, pages 235–246, May 2001.

[7] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B$^+$-Trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the SIGMOD 2002 Conference(to appear)*, June 2002.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proceedings of PLDI '99*, pages 1–12, May 1999.

[9] Z. Cvetanovic and R. E. Kessler. Performance Analysis of the Alpha 21264-Based Compaq ES40 System. In *Proceedings of the 27th ISCA*, pages 192–202, June 2000.

[10] G. Graefe. The Value of Merge-Join and Hash-Join in SQL Server. In *Proceedings of the 25th VLDB*, pages 250–253, Sept. 1999.

[11] G. Graefe and P. Larson. B-tree Indexes and CPU Caches. In *Proceedings of the 17th ICDE Conference*, pages 349–358, April 2001.

[12] J. Gray and G. Graefe. The Five-Minute Rule Ten Years Later. *ACM SIGMOD Record*, 26(4):63–68, Dec. 1997.

[13] IBM Corp. *IBM DB2 Universal Database Administration Guide Version 7*. 2000.

[14] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th VLDB*, pages 294–303, Aug. 1986.

[15] D. Lomet. B-tree Page Size when Caching is Considered. *ACM SIGMOD Record*, 27(3):28–32, Sep. 1998.

[16] D. Lomet. The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. *ACM SIGMOD Record*, 30(3):64–69, Sep. 2001.

[17] S. McFarling. Combining Branch Predictors. Technical Report WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[18] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th VLDB*, pages 78–89, Sept. 1999.

[19] J. Rao and K. A. Ross. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proceedings of the SIGMOD 2000 Conference*, pages 475–486, May 2000.

[20] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.