Improving Index Performance through Prefetching

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

December 2000

CMU-CS-00-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Information Sciences Research Center, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship and by a Faculty Development Award from IBM.

Abstract

In recognition of the crucial role that cache hierarchies play in database performance, recent studies have revisited core database algorithms and data structures in an effort to reduce the number of cache misses. While these efforts to avoid cache misses are certainly helpful, they are not a complete solution for two reasons. First, a large number of cache misses still remain that cannot be eliminated. Second, because modern processors support *prefetching* and other mechanisms to potentially overlap cache misses with computation and other misses, it is not the total *number* of cache misses that dictates performance, but rather the total amount of *exposed miss latency*. Hence an algorithm that is more amenable to prefetching can potentially outperform an algorithm with fewer cache misses.

In this paper, we propose and evaluate *Prefetching B^+-Trees* (pB$^+$-Trees). Such trees are designed to exploit *prefetching* to accelerate two important operations on B$^+$-Tree indices: searches and range scans. To accelerate searches, pB$^+$-Trees use prefetching to effectively create wider nodes than the natural data transfer size: e.g., eight vs. one cache lines or disk pages. These wider nodes reduce the height of the B$^+$-Tree, thereby decreasing the number of expensive misses when going from parent to child without significantly increasing the cost of fetching a given node. Our results show that this technique speeds up search, insertion, and deletion times by a factor of 1.2–1.5 for main-memory B$^+$-Trees. In addition, it outperforms and is complementary to "Cache-Sensitive B$^+$-Trees." To accelerate range scans, pB$^+$-Trees provide arrays of pointers to their leaf nodes. These allow the pB$^+$-Tree to prefetch arbitrarily far ahead, even for nonclustered indices, thereby hiding the normally expensive cache misses associated with traversing the leaves within the range. Our results show that this technique yields over a *sixfold* speedup on range scans of 1000+ keys. Although our experimental evaluation focuses on main memory databases, the techniques that we propose are also applicable to hiding disk latency.

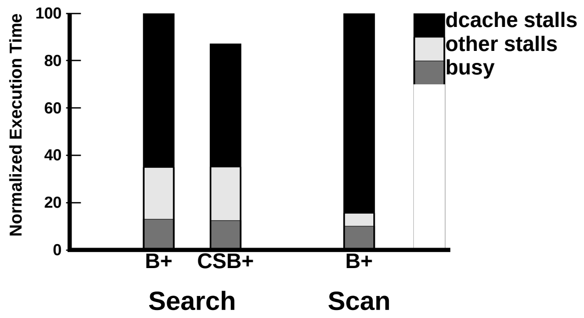


Figure 1: Execution time breakdown for common index accesses (**B+** = B$^+$-Trees, **CSB+** = CSB$^+$-Trees).

1 Introduction

As the gap between processor speed and both DRAM and disk speeds continues to grow exponentially, it is becoming increasingly important to make effective use of caches to achieve high performance on database management systems. Caching exists at multiple levels within modern memory hierarchies: typically two or more levels of SRAM serves as caches for the contents of main memory in DRAM, which in turn is a cache for the contents on disk. While database researchers have historically focused on the importance of this latter form of caching (also known as the "buffer pool"), recent studies have demonstrated that even on traditional disk-oriented databases, roughly 50% or more of execution time is often wasted due to SRAM cache misses [1, 2, 10, 19]. For main-memory databases, it is even clearer that SRAM cache performance is crucial [20]. Hence several recent studies have revisited core database algorithms and data structures in an effort to make them more cache friendly [5, 18, 20, 21, 23].

1.1 Cache Performance of B$^+$-Tree Indices

Index structures are used extensively throughout database systems, and one of the most common implementations of an index is a B$^+$-Tree. While database management systems perform several different operations that involve B$^+$-Tree indices (e.g., single value selections, range selections, and nested loop index joins), these higher-level operations can be decomposed into two key lower-level access patterns: (i) *searching* for a particular key, which involves descending from the root to a leaf node using binary search within a given node to determine which child pointer to follow; and (ii) *scanning* some portion of the index, which involves traversing the leaves through a linked-list structure.[1] While search time is the key factor in single value selections and nested loop index joins, scan time is the dominant effect in range selections.

To illustrate the need for improving the cache performance of both search and scan on B$^+$-Tree indices, Figure 1 shows a breakdown of their simulated performance on a state-of-the-art machine. For the sake of concreteness, we pattern the memory subsystem after the Compaq ES40 [7]—details are provided later in Section 4. The "Search" experiment in Figure 1 looks up 100,000 random keys in a main-memory B$^+$-Tree index after it has been bulkloaded with 10 million keys. The "Scan" experiment performs 100 range scan operations starting at random keys, each of which scans through 1 million ⟨`key`, `tupleID`⟩ pairs retrieving the `tupleID` values. (The results for shorter range scans (e.g., 1000 tuple scans) are similar). The B$^+$-Tree node size is equal to the cache line size, which is 64 bytes.

Each bar in Figure 1 is broken down into three categories: busy, data cache stalls, and other stalls. Both search and scan accesses on B$^+$-Tree indices (the bars labeled "**B+**"—we will explain the "**CSB+**" bar later) spend a significant fraction of their time—65% and 84%, respectively—stalled on data cache misses. Hence there is considerable room for improvement.

[1] For simplicity of exposition, we consider index scans for non-clustered indices, so that the leaves must be traversed. Index scans for clustered indices can directly scan the database table (after searching for the starting key).

1.2 Previous Work on Improving the Cache Performance of Indices

In an effort to improve the cache performance of index *searches* for main-memory databases, Rao and Ross proposed two new types of index structures: "Cache-Sensitive Search Trees" (CSS-Trees) [20] and "Cache-Sensitive B^+-Trees" (CSB^+-Trees) [21]. The premise of their studies is the conventional wisdom that the optimal tree node size is equal to the *natural data transfer size*, which corresponds to the *disk page size* for disk-resident databases and the *cache line size* for main-memory databases. Because cache lines are roughly two orders of magnitude smaller than disk pages (e.g., 64 bytes vs. 4 Kbytes), the resulting index trees for main-memory databases are considerably deeper. Since the number of expensive cache misses is roughly proportional to the height of the tree,[2] it would be desirable to somehow increase the effective fanout (also called the *branching factor*) of the tree, without paying the cost of additional cache misses that this would normally imply.

To accomplish this, Rao and Ross [20, 21] exploit the following insight: by restricting the data layout such that the location of each child node can be directly computed from the parent node's address (or a single pointer), we can eliminate all (or nearly all) of the child pointers. Assuming that keys and pointers are the same size, this effectively doubles the fanout of cache-line-sized tree nodes, thus reducing the height of the tree and the number of cache misses. CSS-Trees [20] eliminate all child pointers, but do not support incremental updates and therefore are only suitable for read-only or OLAP-like environments. CSB^+-Trees [21] do support updates by retaining a single pointer per non-leaf node that points to a contiguous block of its children. Although CSB^+-Trees outperform B^+-Trees on searches, they still perform significantly worse on updates [21] due to the overheads of keeping all children for a given node in sequential order within contiguous memory, especially during node splits.

Returning to the results in Figure 1, the bar labeled "**CSB+**" shows the execution time of CSB^+-Trees (normalized to that of B^+-Trees) for the same index search experiment. As we see in Figure 1, CSB^+-Trees eliminate 20% of the data cache stall time, thus resulting in an overall speedup[3] of 1.15 for searches. While this is a significant improvement, over half of the remaining execution time is still being lost to data cache misses; hence there is significant room for further improvement. In addition, these search-oriented optimizations provide no benefit to *scan* accesses, which are suffering even more from data cache misses.

1.3 Our Approach: *Prefetching B^+-Trees*

Modern microprocessors provide the following mechanisms for coping with large cache miss latencies. First, they allow multiple outstanding cache misses to be in flight simultaneously for the sake of exploiting parallelism within the memory hierarchy. For example, the Compaq ES40 [7] supports 32 in-flight loads, 32 in-flight stores, and eight outstanding off-chip cache misses per processor, and its crossbar memory system supports 24 outstanding cache misses. Second, to help applications take full advantage of this parallelism, they also provide *prefetch* instructions which enable software to move data into the cache before it is needed. Previous studies (which did not target databases specifically) have demonstrated that for both array-based and pointer-based program codes, prefetching can successfully *hide* much of the performance impact of cache misses by overlapping them with computation and other misses [13, 16]. Hence for modern machines, it is not the *number* of cache misses that dictates performance, but rather the amount of *exposed miss latency* that cannot be successfully hidden through techniques such as prefetching.

In this paper, we propose and study *Prefetching B^+-Trees* (pB^+-Trees), which are designed to use prefetching to limit the exposed miss latency. Tree-based indices such as B^+-Trees pose a major challenge for prefetching search and scan accesses, in that both access patterns suffer from the *pointer-chasing problem* [13]: The data dependencies through pointers make it difficult to prefetch sufficiently far ahead to limit the exposed miss latency. For index *searches*, pB^+-Trees seek to reduce this problem by having wider nodes than the natural data transfer size, e.g., eight vs. one cache lines (or disk pages). These wider nodes reduce the height of the tree, thereby decreasing the number of expensive misses when going from parent to child. The key observation is that by using prefetching, the wider nodes come almost for free:

[2] Nodes that are close to the root are likely to remain in the cache if the index is reused repeatedly, but this is a relatively small effect for main-memory indices, given the capacity of the caches and the number of levels in the tree.

[3] Throughout this paper, we report performance improvements in terms of their *speedup factor*, i.e., the original time divided by the improved time.

all the cache lines in a wider node can be fetched about as quickly as the single cache line of a traditional node. To accelerate index *scans*, we introduce arrays of pointers to the B^+-Tree leaf nodes which allow us to prefetch arbitrarily far ahead, thereby hiding the normally expensive cache misses associated with traversing the leaves within the range. Of course, indices may be frequently updated. Perhaps surprisingly, we demonstrate that insertion and deletion times actually *decrease* with our techniques, despite any overheads associated with maintaining the wider nodes and the arrays of pointers.

1.4 Contributions of This Paper

This paper makes the following contributions. First, to our knowledge, this is the first study to explore how prefetching can be used to accelerate search and scan operations on B^+-Tree indices. We propose and study the *Prefetching B^+-Tree* (pB^+-Tree). Second, we demonstrate that contrary to conventional wisdom, the optimal B^+-Tree node size on a modern machine is often *wider* than the natural data transfer size, since we can use prefetching to fetch each piece of the node simultaneously. Our approach offers the following advantages relative to CSB^+-Trees: (i) we achieve better search performance because we can increase the fanout by more than the factor of two that CSB^+-Trees provide, e.g., we can increase it by a factor of eight by making the nodes eight times wider; (ii) we achieve better (rather than worse) performance on updates relative to B^+-Trees, because our improved search speed more than offsets any increase in node split cost due to wider nodes; and (iii) we do not require any fundamental changes to the original B^+-Tree data structures or algorithms. In addition, we find that our approach is *complementary* to CSB^+-Trees, in that both techniques can be used together to advantage. Third, we demonstrate how the pB^+-Tree can effectively hide over 90% of the cache miss latency suffered by (non-clustered) index scans, thus resulting in *a factor of 6.5–8.7 speedup* over a range of scan lengths. While our experimental evaluation is performed within the context of main memory databases, we believe that our techniques are also applicable to hiding disk latency, in which case the prefetches will move data from disk into main memory.

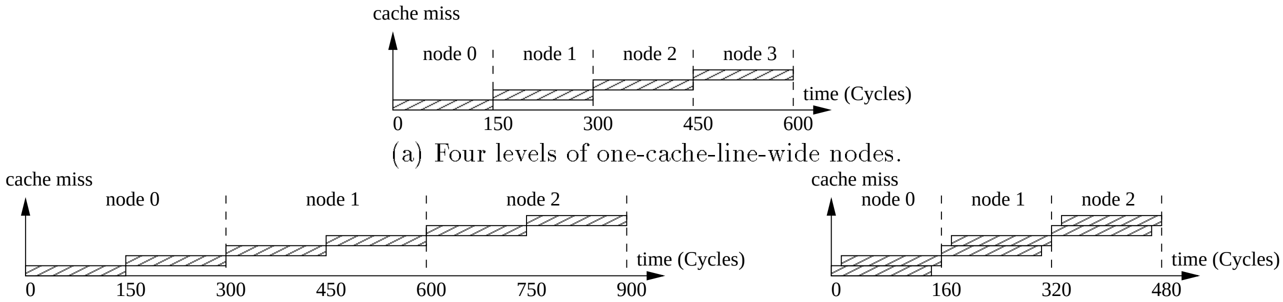
The remainder of this paper is organized as follows. In Sections 2 and 3, we discuss how pB^+-Trees use prefetching to accelerate index searches and scans, respectively. To quantify the benefits of these techniques, we present experimental results in Section 4. We discuss further issues related to B^+-Tree operations in Section 5, and finally we conclude in Section 6.

2 Index Searches: Using Prefetching to Create Wider Nodes

Recall that during a B^+-Tree search, we start from the root, performing a binary search in each non-leaf node to determine which child to visit next. Upon reaching a leaf node, a final binary search returns the key position (or the preceding key position in the case of an insertion). Regarding the cache behavior, we expect at least one expensive cache miss to occur each time we move down a level in the tree. Hence the number of cache misses is roughly proportional to the height of the tree (minus any nodes that might remain in the cache if the index is reused). Thus, having wider tree nodes for the sake of reducing the height of the tree might seem like a good idea. Unfortunately, in the absence of prefetching (i.e., when all cache misses are equally expensive and cannot be overlapped), making the tree nodes wider than the *natural data transfer size*—i.e., a cache line for main-memory databases (and a disk page for disk-resident databases)—actually *hurts* performance rather than helps it, as has been shown in previous studies [20, 21]. The reason for this is that the number of additional cache misses at each node more than offsets the benefits of reducing the number of levels in the tree.

As a small example, consider a main-memory B^+-Tree holding 1000 keys where the cache line size is 64 bytes and the `keys`, child pointers, and `tupleIDs` are all four bytes. If we limit the node size to one cache line, then the B^+-Tree will contain at least four levels. Figure 2(a) illustrates the resulting cache behavior, where the four cache misses would cost a total of 600 cycles on our Compaq ES40-based machine model [7]. If we double the node width to *two* cache lines, the height of the B^+-Tree can be reduced to *three* levels. However, as we see in Figure 2(b), this would result in *six* cache misses, thus increasing execution time by 50%. In general, if w is the number of cache lines per node and m is the number of pointers per one-cache-line node,[4]

[4] Throughout this paper, we consider for simplicity *fixed-size* `keys`, `tupleIDs`, and pointers. Thus m is the same for all nodes. We also assume that `tupleIDs` and pointers are the same size.



(a) Four levels of one-cache-line-wide nodes.

(b) Three levels of two-cache-lines-wide nodes.

(c) Part (b) with prefetching.

Figure 2: Cache behaviors of various B$^+$-Tree searches. The cache miss latency is 150 cycles to main memory, and a subsequent access can begin 10 cycles later; these latencies are based on the Compaq ES40 [7].

then the number of cache misses in a B$^+$-Tree of N `tupleID`s is roughly proportional to

$$\textit{cache lines per node} \times \textit{levels in tree} = w \times \left\lceil \log_{wm}\left(\frac{N}{wm-1}\right) + 1 \right\rceil \tag{1}$$

To see this, observe that each non-leaf node contains wm child pointers, so that the fanout is wm, and each leaf node holds $wm - 1$ $\langle$`key`, `tupleID`$\rangle$ pairs. Although the number of levels decreases as w increases, w, which is the dominating factor in the equation, gets worse. Therefore equation (1) is minimized when $w = 1$.

With prefetching, however, the premise that all cache misses are equally costly is no longer valid, for it becomes possible to *hide* the latency of any miss whose address can be predicted sufficiently early. Returning to our example, if we prefetch the second half of each two-cache-line-wide tree node so that it is fetched in parallel with the first half—as illustrated in Figure 2(c)—we can achieve significantly *better* (rather than worse) performance compared with the one-cache-line-wide nodes in Figure 2(a). The extent to which the misses can be overlapped depends upon the implementation details of the memory hierarchy, but the trend is toward supporting greater parallelism. In fact, with multiple cache and memory banks and crossbar interconnects, it is possible to completely overlap multiple cache misses. Figure 2(c) illustrates the timing on our Compaq ES40-based machine model, where back-to-back misses to memory can be serviced once every 10 cycles, which is a small fraction of the overall 150 cycle miss latency. Therefore even without perfect overlap of the misses, we can still potentially achieve large performance gains (a speedup of 1.25 in this example) by creating wider than normal B$^+$-Tree nodes.

Hence the first aspect of our pB$^+$-Tree design is to use prefetching to "create" nodes that are *wider* than the natural data transfer size, but where the entire miss penalty for each extra-wide node is comparable to that of an original B$^+$-Tree node.

2.1 Modifications to the B$^+$-Tree Algorithm

We consider a standard B$^+$-Tree node structure: Each *non-leaf* node is comprised of some number, $d \gg 1$, of `childptr` fields, $d - 1$ `key` fields, and one `keynum` field that records the number of `key`s stored in the node (at most $d - 1$). (All notation is summarized in Table 1.) Each *leaf* node is comprised of $d - 1$ `key` fields, $d - 1$ associated `tupleID` fields, one `keynum` field, and one `next-leaf` field that points to the next leaf node in key order. Our first modification is to store the `keynum` and all of the `key`s prior to any of the pointers or `tupleID`s in a node. This simple layout optimization allows the binary search to proceed without waiting to fetch all the pointers. Our search algorithm is a straightforward extension of the standard B$^+$-Tree algorithm, and we now describe only the parts that change.

Search: Before starting a binary search, we prefetch all of the cache lines that comprise the node.

Insertion: Since an index search is first performed to locate the position for insertion, all of the nodes on the path from the root to the leaf are already in the cache before the real insertion phase. The only additional cache misses are caused by newly allocated nodes, which we prefetch in their entirety before redistributing the keys.

Table 1: Variable names and terminology used throughout this paper.

Variable	Definition
w	number of cache lines in an index node
m	number of child pointers in a one-cache-line-wide index node
N	number of $\langle$`key`, `tupleID`$\rangle$ pairs in an index
d	number of child pointers in a non-leaf B^+-Tree node ($= w \times m$)
T_1	full latency of a cache miss
T_{next}	incremental latency of an additional pipelined cache miss
B	normalized memory bandwidth $\left(B = \frac{T_1}{T_{\text{next}}}\right)$
k	prefetching distance (number of nodes to prefetch ahead)
c	chunk size (number of cache lines in a jump-pointer array chunk)
$p^w B^+$-Tree	pB^+-Tree with w-line-wide nodes and no jump-pointer arrays
$p_e^w B^+$-Tree	pB^+-Tree with w-line-wide nodes and *external* jump-pointer arrays
$p_i^w B^+$-Tree	pB^+-Tree with w-line-wide nodes and *internal* jump-pointer arrays

Deletion: We perform *lazy deletion* as in Rao and Ross [21]. If more than one key is in the node, we simply delete the key. It is only when the last key in a node is deleted that we try to redistribute keys or delete the node. Since index search is also performed prior to deletion, the entire root-to-leaf path is in the cache. Key redistribution is the only potential cause of additional misses; hence when all keys in a node are deleted, we prefetch the sibling node from which keys will be redistributed.

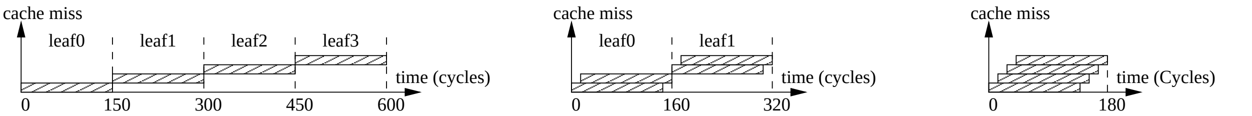
Prefetching can also be used to accelerate the *bulkload* of a B^+-Tree. However, because this is expected to occur infrequently, we focus instead on the more frequent operations of search, insertion and deletion.

2.2 Qualitative Analysis

As discussed earlier in this section, we expect search times to improve through our scheme because it reduces the number of levels in the B^+-Tree without significantly increasing the cost of accessing each level. What about the performance impact on updates? Updates always begin with a search phase, which will be sped up. The expensive operations only occur either when the node becomes too full upon an insertion and must be split, or when a node becomes empty upon a deletion and keys must be redistributed. Although node splits and key redistributions are more costly with larger nodes, the relative frequency of these expensive events should decrease. Therefore we expect update performance to be comparable to, or perhaps even better than, B^+-Trees with single-line nodes. Note that this would be in contrast with CSB^+-Trees, where update performance is generally worse than B^+-Trees [21].

The space overhead of the index is strictly reduced with wider nodes. There is a slight improvement in the amount of storage for leaf nodes, because the fixed overhead per leaf node (i.e., the `keynum` and `next-leaf` fields) is amortized across a larger number of $\langle$`key`, `tupleID`$\rangle$ pairs. The more significant effect is the reduction in the number of non-leaf nodes. For a full tree, each leaf node contains $d-1$ $\langle$`key`, `tupleID`$\rangle$ pairs. The number of non-leaf nodes is dominated by the number of nodes in the level immediately above the leaf nodes, and hence is approximately $\frac{N}{d(d-1)}$. As the fanout d increases with wider nodes, the node size grows linearly but the number of nodes decreases quadratically, resulting in a near linear decrease in the non-leaf space overhead.

Finally, an interesting consideration is to determine the optimal node size, given prefetching. Should nodes simply be as wide as possible? There are two system parameters that affect this answer. The first is the extent to which the memory subsystem can overlap multiple cache misses. We quantify this as the latency of a full cache miss (T_1) divided by the additional time until another pipelined cache miss would also complete (T_{next}). We call this ratio (i.e., $\frac{T_1}{T_{\text{next}}}$) the *normalized bandwidth* (B). For example, in our Compaq ES40-based machine model, $T_1 = 150$ cycles, $T_{\text{next}} = 10$ cycles, and hence $B = 15$. The larger the value of B, the greater the system's ability to overlap parallel accesses, and hence the greater likelihood of benefiting from wider index nodes. In general, we do not expect the optimal number of cache lines per node (w_{optimal})



(a) Scanning 4 one-line nodes (b) Scanning 2 two-line nodes (c) Scanning with prefetching

Figure 3: Cache behaviors of index range scans.

to exceed B, since beyond that point we could have completed a binary search and moved down to the next level in the tree. The second system parameter that potentially limits the optimal node size is the size of the cache, although in practice this does not appear to be a limitation given realistic values of B.

Let us now consider a more quantitative analysis of the optimal node width (w_{optimal}). A pB$^+$-Tree with N $\langle$`key`, `tupleID`$\rangle$ pairs contains at least $\left\lceil \log_d \left(\frac{N}{d-1}\right) + 1 \right\rceil$ levels. With our data layout optimization of putting keys before child pointers, $\frac{3}{4}$ of the node is read on average. Hence the average memory stall time for a search in a full tree is

$$\left\lceil \log_d \frac{N}{d-1} + 1 \right\rceil \times \left(T_1 + \left(\left\lceil \frac{3w}{4} \right\rceil - 1 \right) \times T_{\text{next}} \right) = T_{\text{next}} \times \left\lceil \log_{wm} \frac{N}{wm-1} + 1 \right\rceil \times \left(B + \left\lceil \frac{3w}{4} \right\rceil - 1 \right) \quad (2)$$

By computing the value of w that minimizes this cost, we can find w_{optimal}. For example, in our simulations where $m = 8$ and $B = 15$, then by averaging over tree sizes $N = 10^3, \ldots, 10^9$, we can compute from equation (2) that $w_{\text{optimal}} = 8$. If the memory subsystem bandwidth increases such that $B = 50$, then w_{optimal} increases to 22.

In summary, comparing our pB$^+$-Trees with conventional B$^+$-Trees, we expect better search performance, comparable or somewhat better update performance, and lower space overhead. Having addressed index search performance, we now turn our attention to index range scans.

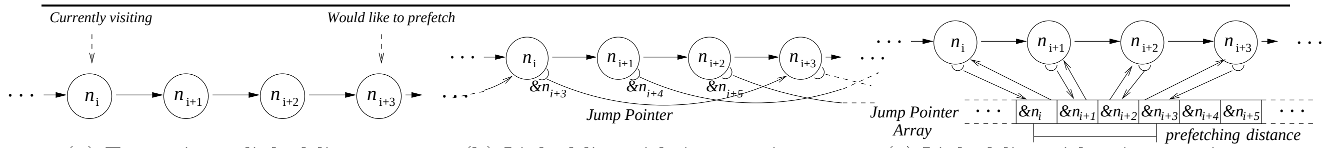
3 Index Scans: Prefetching Ahead Using Jump-Pointer Arrays

Recall that an index range scan takes starting and ending keys as arguments and returns a list of either the `tupleID`s or the tuples themselves with keys that fall within this range. The first step in performing a range scan is to *search* the B$^+$-Tree to find the starting leaf node. Once the starting leaf is known, the scan then follows the `next-leaf` pointers, visiting the leaf nodes in order. As the scan proceeds, the `tupleID`s (or tuples) are copied into a return buffer. This process stops when either the ending key is found or the return buffer fills up. In the latter case, the scan procedure pauses and returns the buffer to the caller (often a join node in a query execution plan), which then consumes the data and resumes the scan where it left off. Hence a range selection involves one key search and often multiple leaf node scan calls. Throughout this section, we will focus on range selections that return `tupleID`s, although returning the tuples themselves (or other variations) is a straightforward extension of our algorithm, as we will discuss later in Section 5.

As we saw already in Figure 1, the cache performance of range scans is abysmal: 84% of execution time is being lost to data cache misses in that experiment. Figure 3(a) illustrates the problem, which is that the full cache miss latency is suffered for each leaf node. A partial solution is to use the technique described in Section 2: If we make the leaf nodes multiple cache lines wide and prefetch each component of a leaf node in parallel, we can reduce the frequency of expensive cache misses, as illustrated in Figure 3(b). While this is helpful, our goal is to *fully* hide the miss latencies, as illustrated in Figure 3(c). In order to do that, we must first overcome the *pointer-chasing problem*.

3.1 Solving the Pointer-Chasing Problem

Figure 4(a) illustrates the *pointer-chasing problem*, which was observed by Luk and Mowry [13, 14] in the context of prefetching pointer-linked data structures (i.e., linked-lists, trees, etc.) in general-purpose applications. Assuming that three nodes worth of computation are needed to hide the miss latency, then when node n_i in Figure 4(a) is visited, we would like to be launching a prefetch of node n_{i+3}. To compute the



(a) Traversing a linked list (b) Linked list with jump pointers (c) Linked list with a jump-pointer array

Figure 4: Illustration of the pointer-chasing problem and how it can be addressed.

address of node n_{i+3}, we would normally follow the pointer chain through nodes n_{i+1} and n_{i+2}. However, this would incur the full miss latency to fetch n_{i+1} and then to fetch n_{i+2}, before the prefetch of n_{i+3} could be launched, thereby defeating our goal of hiding the miss latency of n_{i+3}.

Luk and Mowry proposed three solutions to the pointer-chasing problem [13, 14], two of which are applicable to linked lists. The first scheme (*data-linearization prefetching*) involves arranging the nodes in memory such that their addresses can be trivially calculated without dereferencing any pointers. For example, if the leaf nodes of the B$^+$-Tree are arranged sequentially in contiguous memory, they would be trivial to prefetch. However, this will only work in read-only situations, and we would like to support frequent updates. The second scheme (*history-pointer prefetching*) involves creating new pointers—called *jump pointers*—which point from a node to the node that it should prefetch. For example, Figure 4(b) shows how node n_i could directly prefetch node n_{i+3} using three-ahead jump pointers.

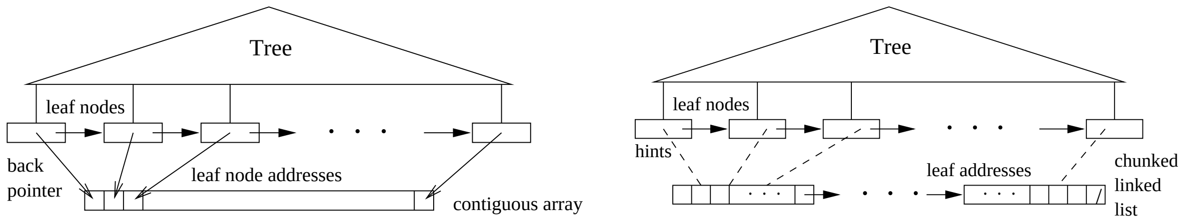
In our study, we will build upon the concept of jump pointers, although we will customize them to the specific needs of B$^+$-Tree indices. For example, rather than storing jump pointers directly in the leaf nodes, we instead pull them out into a separate array, which we call the *jump-pointer array*, as illustrated in Figure 4(c). To initiate prefetching with a jump-pointer array, the starting leaf node uses a pointer to locate its position within the array; it then adjusts its offset within the array based on the desired prefetching distance to find the appropriate leaf node address to prefetch. As the scan proceeds, the prefetching task simply continues to walk ahead in the jump-pointer array (which itself is prefetched) without having to dereference the actual leaf nodes again. In addition to this advantage, jump-pointer arrays offer other benefits relative to storing the jump pointers directly in the leaf nodes. For example, we can adjust the prefetching distance without changing either the jump-pointer array or the B$^+$-Tree nodes by simply changing the offset used within the array. This allows us to dynamically adapt to changing performance conditions on a given machine, or if the code migrates to different machines. In addition, the same jump-pointer array can be reused to target different latencies in the memory hierarchy (e.g., disk latency vs. memory latency). Other advantages include improved robustness and more relaxed concurrency control on the jump-pointer arrays, since we can treat the pointers simply as hints, etc.

Jump-pointer arrays do introduce additional space overhead, although that overhead is relatively small since the array only contains one pointer per leaf node. For example, for the 8-cache-line-wide nodes that we use later in our experiments, the jump-pointer array is only 0.8% of the size of the leaf nodes themselves. Nevertheless, it is possible to reduce this space overhead even further by reusing internal nodes within the B$^+$-Tree, as we will discuss later in Section 3.5.

From an abstract point of view, one might think of the jump-pointer array as a single large, contiguous array, as illustrated in Figure 5(a). (For simplicity, we omit the pointers from jump-pointer array back to the leaf nodes in this figure.) If updates did not occur, it would be easy to implement the jump-pointer array this way in practice. However, in a read-only situation there is no motivation for using a jump-pointer array, since we could simply arrange the leaf nodes themselves contiguously and use *data-linearization prefetching* [13, 14], as we mentioned earlier. Hence a key issue in implementing jump-pointer arrays is that they must handle updates gracefully.

3.2 Implementing Jump-Pointer Arrays to Support Efficient Updates

Let us briefly consider the problems created by updates if we attempted to maintain the jump-pointer array as a single contiguous array as shown in Figure 5(a). If a leaf node is deleted, then a gap will be created within the array unless it is compacted. Since compacting an array of such a large size would be quite expensive, it is more likely that we would simply leave the hole in the short term, and perhaps perform compaction of



(a) single array with real back-pointers (b) chunked linked list with "hint" back-pointers

Figure 5: Structures to implement external jump-pointer arrays.

the entire array periodically but infrequently. The only real problem with a hole in the jump-pointer array is that we need to skip over it to get to the next real prefetching address, although one can easily imagine working around this issue. The more significant problem occurs when new leaves are *inserted*, in which case we would like to create a hole in the appropriate position for the new jump pointer. If no nearby holes could be located, then this could potentially involve copying a very large amount of data within the array in order to create the hole. In addition, for each jump-pointer that moves within the array, the corresponding back-pointer from the leaf node into the array would also need to be updated, which could be very costly. Clearly we would not want to pay such a high cost upon updates.

To avoid these problems, we improve upon the naive contiguous array implementation of jump-pointer arrays in the following three ways. First, we break the contiguous array into a chunked linked list—as illustrated in Figure 5(b)—which allows us to limit the impact of an update to its corresponding chunk. The chunk size (c, in units of cache lines) is chosen to be sufficiently large relative to the cache miss latency to ensure that we only need to prefetch one chunk ahead in order to fully hide the miss latency of the chunks themselves. As we will see later in our experimental results, we achieve excellent performance on both scans and updates using this approach.

Second, we actively attempt to interleave empty slots within the jump-pointer array so that insertions can proceed quickly. During bulkload or when a chunk splits, the jump pointers are stored such that empty slots are evenly distributed to maximize the chance of finding a nearby empty slot for insertion. When a jump-pointer is deleted, we simply leave an empty slot in the chunk.

Finally, we alter the meaning of the pointer from a leaf node to its position in the jump-pointer array such that it is merely a hint. The pointer should point to the correct chunk, but the position within that chunk may be imprecise. The advantage of this is that upon insertion, we do not need to update hints pointing to any addresses moved in order to create a hole. We only update a `hint` field when: (i) the precise position of its jump-pointer array position is looked up during range scan or insertion, in which case the leaf node should be already in cache and updating the `hint` is almost free; and (ii) when a chunk splits and addresses are redistributed, in which case we are forced to update the `hint`s to point to the new chunk. The cost of using hints, of course, is that we need to search for the correct location within the chunk in some cases. In practice, however, the hints appear to be good approximations of the true positions, and searching for the precise location is not a costly operation (e.g., it should not incur any cache misses).

In summary, the net effect of these three enhancements is that nothing moves during deletions, only a small number of jump pointers (between the insertion position and the nearest empty slot) typically move during insertions, and in neither case do we normally update the `hint`s within the leaf nodes. Thus we expect jump-pointer arrays to perform well during updates. Having described the data structure that we use to facilitate prefetching, we now describe our prefetching algorithm itself.

3.3 Prefetching Algorithm

Recall that the basic range scan algorithm consists of a loop that visits a leaf on each iteration by following a `next-leaf` pointer. To augment this algorithm to support prefetching, we add prefetches both prior to this loop (for the *startup* phase), and inside the loop (for the stable *steady-state* phase). Let k be the desired *prefetching distance*, in units of leaf nodes (we discuss below how to select k). During the startup phase, we perform range prefetching for the first k leaf nodes.[5] Note that these prefetches proceed in parallel, exploiting the available memory hierarchy bandwidth. During each loop iteration (i.e. in the steady-state phase), prior to visiting the current leaf node in the range scan, we range prefetch the leaf node that is k nodes after the current leaf node. The goal is to ensure that by the time the basic range scan loop is ready to visit a leaf node, that node is already in the cache, due to our earlier prefetching. With this framework in mind, we now describe further details of our algorithm.

First, when the range scan begins, we must find the location of the starting leaf within the jump-pointer array. To do this, we follow the `hint` pointer from the starting leaf node to see whether it is a precise match—i.e. whether the `hint` points to a pointer back to the starting leaf. If not, then we start searching within the chunk in both directions relative to the hint position until the matching position is found. As discussed earlier, we expect the distance between the hint and the actual position to be small.

Second, we need to prefetch the jump-pointer array chunks as well as the leaf nodes, and also handle empty slots in the chunks. During the startup phase, we prefetch both the current chunk and the next chunk. We test for and skip all empty slots when looking for a jump pointer. If we come to the end of the current chunk, we will go to the next chunk to get the first non-empty jump-pointer (there is at least one non-empty jump pointer or the chunk should have been deleted). We then prefetch the next chunk ahead in the jump-pointer array. Because we always prefetch the next chunk before prefetching any leaf nodes associated with the current chunk, the next chunk should be in the cache by the time we access it.

Third, although the actual number of `tupleIDs` in the leaf node is unknown when we do range prefetching, we will assume that the leaf is full and prefetch the return buffer area accordingly. Thus the return buffer will always be prefetched sufficiently early.

Finally, we discuss how to select the prefetching distance and the chunk size. The *prefetching distance* (k, in units of nodes to prefetch ahead) is selected as follows. Normally one would derive this quantity by dividing the expected worst-case miss latency by the time it takes to consume one leaf node (similar to what has been done in other contexts [16]). However, because the computation time associated with performing the binary search is expected to be quite small relative to the miss latency, we will assume that the limiting factor is the memory bandwidth. Roughly speaking, we can estimate this bandwidth-limited prefetching distance as

$$k = \left\lceil \frac{B}{w} \right\rceil, \tag{3}$$

where B is the normalized memory bandwidth (recall that $B = \frac{T_1}{T_{next}}$, the number of cache misses that can be in flight simultaneously), and w is the number of cache lines per leaf node. In practice, there is no problem with increasing k a bit to create some extra slack, because any prefetches that cannot proceed are simply buffered within the memory system. Indeed, our experimental results in Section 4 show that the performance is not particularly sensitive to increasing k beyond $\lceil \frac{B}{w} \rceil$.

Regarding the *chunk size* (c), we observe a similar effect (also shown later in Section 4) where increasing it beyond its minimum value has little impact on performance. Therefore, rather than trying to compute c precisely, we will instead focus on estimating the minimum value of c that will be effective. The major factor that dictates the minimum chunk size is that the chunks must be sufficiently large to ensure that we only need to prefetch one chunk ahead to fully hide the miss latency of accessing the chunks themselves. Recall that during the steady-state phase of a range scan, when we get to a new chunk, we immediately prefetch the next chunk ahead so that we can overlap its fetch time with the time it takes to prefetch the leaf nodes associated with the current chunk. Since the memory hierarchy only has enough bandwidth to initiate B cache misses during the time it takes one cache miss to complete, the chunks would clearly be large enough to hide the latency of fetching the next chunk (even if we are bandwidth-limited) if they contained at least B leaf pointers. Since c is in units of cache lines, we compute its value by dividing B by the expected number of leaf pointers within a cache-line-sized portion of a chunk. For a full tree with no empty leaf slots and no empty chunk slots, each cache line can hold $2m$ leaf pointers (since there are only pointers and no keys), in

[5]Note that we need to prefetch not only a leaf node but also the buffer area to hold the resulting `tupleIDs`; to simplify this discussion, when we refer to "range prefetching a leaf node," we mean prefetching the cache lines for both the leaf node and the buffer area where the `tupleIDs` are to be stored.

which case we can estimate the minimum chunk size as

$$c = \left\lceil \frac{B}{2m} \right\rceil . \tag{4}$$

To account for empty chunk slots, we can multiply the denominator in equation (4) by the fraction of chunk slots that are expected to be full (a value similar to the *bulkload factor*), which would increase c somewhat. A second factor that could (in theory) dictate the minimum chunk size is that each chunk should contain at least k leaf pointers so that our prefetching algorithm can get sufficiently far ahead. However, since $k \leq B$ from equation (3), the chunk size in equation (4) should be sufficient. As we mentioned earlier, increasing c beyond this minimum value to create some extra slack for empty leaf nodes and empty chunk slots does not hurt performance in practice, as our experimental results demonstrate later in Section 4.

3.4 Summary and Qualitative Analysis

We now briefly summarize the various aspects of our algorithm (referring the reader to Table 1, as needed):

Range Scan: Given the starting leaf node, we follow the `hint` in the leaf to its jump-pointer array position, and locate the precise location for that leaf in the chunk. (At this point we update the `hint` for free.) In the start-up phase, we perform range prefetching for k nodes. In each range scan loop iteration, we prefetch the kth leaf node ahead of the current one.

Insertion: If the insertion does not result in a leaf node split, then there is no updating to be done. In the infrequent case that a leaf node splits, if the jump-pointer array chunk has empty slots, the precise location of the jump pointer is located as well as the nearest empty slot. The jump pointers between these two locations are then moved and the new jump pointer is inserted. The `hint` in the new leaf node is set, and the `hint` in the original leaf node is updated (if needed). On the other hand, in the rare case that a leaf node splits *and* the chunk is full, then a new chunk is allocated and jump pointers are redistributed so that empty slots are evenly distributed throughout the two chunks. The `hint` in the new leaf node is set, and the `hint`s in all leaf nodes with redistributed jump pointers are updated.

Deletion: If the deletion does not empty a leaf node, then there is no updating to be done. In the rare case that a leaf node is emptied, and therefore is being deleted, the precise location in the jump-pointer array is determined. If the chunk contains at least two addresses, then the jump pointer slot is simply set to null, indicating an empty slot. Otherwise, the chunk is removed from the jump-pointer array.

Search: The search algorithm is unchanged. However, because each leaf node now contains a `hint` field, the maximum number of ⟨`key`, `tupleID`⟩ pairs in a leaf is reduced by one (from $d-1$ to $d-2$).

Bulkload: The bulkload algorithm is extended to allocate and initialize the jump-pointer array, filling it with pointers to the leaf nodes. The bulkload factor determines the number of occupied slots in each jump-pointer array chunk, just as it determines the number of occupied slots in leaf nodes.

Given sufficient memory system bandwidth, our prefetching scheme hides the full memory latency experienced at every leaf node visited during range scan operations. As discussed earlier, we also expect good performance on updates. However, there is a space overhead associated with the jump-pointer array in order to store the $\frac{N}{d-2}$ jump pointers. Given our technique described earlier in Section 2 for creating wider B^+-Tree nodes, the resulting increase in the fanout (d) will help reduce this overhead. However, if this space overhead is still a concern, we now describe how it can be reduced further.

3.5 Saving Space through *Internal* Jump-Pointer Arrays

So far we have described how a jump-pointer array can be implemented by creating a new *external* structure to store the jump pointers (as illustrated earlier in Figure 5). However, there is an existing structure *within* a B^+-Tree that already contains pointers to the leaf nodes, namely, the *parents* of the leaf nodes. We will refer to these parent nodes as the *bottom non-leaf nodes*. The child pointers within a bottom non-leaf node

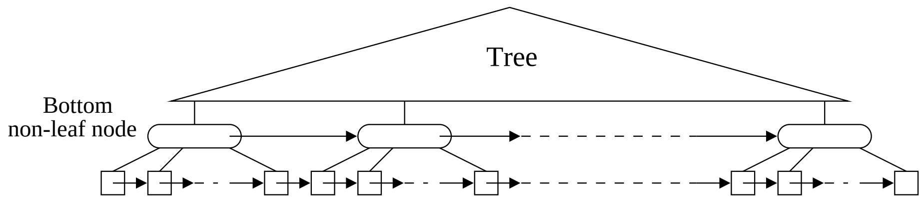


Figure 6: Adding next pointers to bottom non-leaf nodes

correspond to the jump-pointers within a chunk of the external jump-pointer array described in Section 3.2. A key difference, however, is that there is no easy way to traverse these bottom non-leaf nodes quickly enough to perform prefetching. A potential solution is to connect these bottom non-leaf nodes together in leaf key order using linked-list pointers. (Note that this is sometimes done already for concurrency control purposes [24].)

Figure 6 illustrates how this might be accomplished to create what is effectively an *internal* jump-pointer array. Note that in the figure, leaf nodes do not contain pointers to their position within the jump-pointer array. It turns out that such pointers are not necessary for this internal implementation, because that position will be determined during the binary search of the leaf node's parent. If we simply record or pass this position along to the leaf node when we descend to it, then the prefetching can begin appropriately.

Maintaining a next pointer for a bottom non-leaf node is fairly similar to maintaining a next pointer for a leaf node. Therefore bulkload, insertion and deletion algorithms can be easily obtained by extending the existing B^+-Tree algorithms. The search algorithm is the same, with the only difference being that the maximum number of pointers in a bottom non-leaf node is reduced by one. The prefetching algorithm for range scan is similar to the one described earlier for external jump-pointer arrays, although we do not need to locate the starting leaf within the jump-pointer array (because the position has already been recorded, as discussed above).

This approach is attractive with respect to space overhead, since the only overhead is one additional pointer per bottom non-leaf node. The overhead of updating this pointer should be insignificant, because it only needs to be changed in the rare event that a bottom non-leaf node splits or is deleted. One potential limitation of this approach, however, is that the length of a "chunk" in this jump-pointer array is dictated by the B^+-Tree structure, and may not be easily adjusted to satisfy large prefetch distance requirements (e.g., for hiding disk latencies).

In the remainder of this paper, we will use the notations "p_eB^+-Tree" and "p_iB^+-Tree" to refer to pB^+-Trees with *external* and *internal* jump-pointer array structures, respectively.

4 Experimental Results

To facilitate comparisons with CSB^+-Trees, we present our experimental results in a *main-memory* database environment. We begin by describing the framework for the experiments, including our performance simulator and the implementation details of the index structures that we compare. The three subsections that follow present our experimental results for index search, index scan, and updates. Next, we present results for "mature" trees, followed by sensitivity analysis. Finally, we present a detailed cache performance study for a few of our earlier experiments.

4.1 Experimental Framework

4.1.1 Machine Model

We evaluate the performance impact of *Prefetching B^+-Trees* through detailed simulations of fully-functional executables running on a state-of-the-art machine. Since the gap between processor and memory speeds is continuing to increase dramatically with each new generation of machines, it is important to focus on the performance characteristics of machines in the near future rather than in the past. Hence we base

Table 2: 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 [15]

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

our memory hierarchy on the Compaq ES40 [7] (one of the most advanced computer systems announced to date), but we update it slightly to include a dynamically-scheduled, superscalar processor similar to the MIPS R10000 [25] running at a clock rate of 1 GHz. The simulator performs a cycle-by-cycle simulation, modeling the rich details of the processor including the pipeline, register renaming, the reorder buffer, branch prediction, branching penalties, the memory hierarchy (including all forms of contention), etc. Table 2 shows the key parameters of the simulator.

Given the parameters in Table 2, one can see that the normalized memory bandwidth (B)—i.e. the number of cache misses to memory that can be serviced simultaneously—is 15, since

$$B = \frac{T_1}{T_{\text{next}}} = \frac{150}{10} = 15. \tag{5}$$

This is slightly pessimistic compared with the actual Compaq ES40 [7], where $B = 16.25$, and is intended to reflect other recent memory system designs [3]. As shown later in Section 4.6, small variations in B do not substantively alter the results of our studies.

We compiled our C source code into MIPS executables using version 2.95.2 of the gcc compiler with optimization flags enabled. We added prefetch instructions to the source code by hand, using the gcc `ASM` macro to translate these directly into valid MIPS prefetch instructions.

4.1.2 B$^+$-Trees Studied and Implementation Details

Our experimental study compares pB$^+$-Trees of various node widths w with B$^+$-Trees and CSB$^+$-Trees. We consider both p_e^wB$^+$-Trees and p_i^wB$^+$-Trees (described earlier in Sections 3.2–3.4 and Section 3.5, respectively). We also consider the combination of both pB$^+$-Tree and CSB$^+$-Tree techniques, which we denote as a pCSB$^+$-Tree.

We implemented bulkload, search, insertion, deletion, and range scan operations for: (i) standard B$^+$-Trees; (ii) p^wB$^+$-Trees for node widths $w = 2, 4, 8$, and 16; (iii) p_e^8B$^+$-Trees; and (iv) p_i^8B$^+$-Trees. For these latter two cases, the node width $w = 8$ was selected because our experiments showed that this choice resulted in the best search performance (consistent with the analytical computation in Section 2). We also implemented bulkload and search for CSB$^+$-Trees and pCSB$^+$-Trees. Although we did not implement insertion or deletion for CSB$^+$-Trees, we conduct the same experiments as in Rao and Ross [21] (albeit in a different memory hierarchy) to facilitate a comparison of the results.[6] Although Rao and Ross present techniques to improve CSB$^+$-Tree search performance *within* a node [21], we only implemented standard binary search for all the trees studied because our focus is on memory performance (which is the primary bottleneck, as shown earlier in Figure 1).

Our pB$^+$-Tree techniques improve performance over a range of `key`, pointer, and `tupleID` sizes. For concreteness, we report experimental results where the `key`s, pointers, and `tupleID`s are 4 bytes each, as was done in previous studies [20, 21]. As discussed in Section 2, we use a standard B$^+$-Tree node structure, consistent with previous studies. For the B$^+$-Tree, each node is one cache line wide (i.e., 64 bytes). Each

[6] Note that direct comparisons are not possible due to the different memory hierarchies, although we would expect the same general trends to apply.

non-leaf node contains a **keynum** field, 7 **key** fields and 8 **childptr** fields, while each leaf node contains a **keynum** field, 7 **key** fields, 7 associated **tupleID** fields, and a **next-leaf** pointer.[7] The nodes of the pB$^+$-Trees are the same as the B$^+$-Trees, except that they are wider. So for eight-cache-line-wide nodes, each non-leaf node is 512 bytes and contains a **keynum** field, 63 **key** fields, and 64 **childptr** fields, while each leaf node contains a **keynum** field, 63 **key** fields, 63 associated **tupleID** fields, and a **next-leaf** pointer. For the p^8_eB$^+$-Tree, non-leaf nodes have the same structure as for the pB$^+$-Tree, while each leaf node has a **hint** field and one fewer **key** and **tupleID** fields. For the p^8_iB$^+$-Tree, the only difference from the pB$^+$-Tree is its bottom non-leaf nodes: each bottom non-leaf node has a **next-sibling** pointer, and one fewer **key** and **childptr** fields. For the CSB$^+$-Tree and the pCSB$^+$-Tree, each non-leaf node has only one **childptr** field. For example, a CSB$^+$-Tree non-leaf node has a **keynum** field, 14 **key** fields, and a **childptr** field. All tree nodes are aligned on a 64 byte boundary when allocated.

For the p^8_eB$^+$-Tree and p^8_iB$^+$-Tree experiments, we need to select the prefetch distance (for both) and the chunk size (for the former). According to equations (3) and (5), we should select $k = \lceil \frac{B}{w} \rceil = \lceil \frac{15}{8} \rceil = 2$. However, as discussed in Section 3, it is often advantageous to slightly increase k in order to create some extra slack. We set $k = 3$, to create extra slack for the prefetching of chunks and non-leaf nodes. (Our sensitivity analysis in Section 4.6 will show that selecting $k = 2$, 3, or 4 results in similar scan performance.) As for the chunk size, according to equation (4) and the discussion that follows, we should select c to be at least $\lceil \frac{B}{2m} \rceil = \lceil \frac{15}{16} \rceil = 1$. We conservatively select $c = 8$, i.e., each chunk is 8 cache lines wide, so that each chunk contains 126 leaf pointer fields. (Our sensitivity analysis in Section 4.6 will show that selecting $c = 1, 2, \ldots, 32$ results in similar scan performance.)

4.2 Search Performance

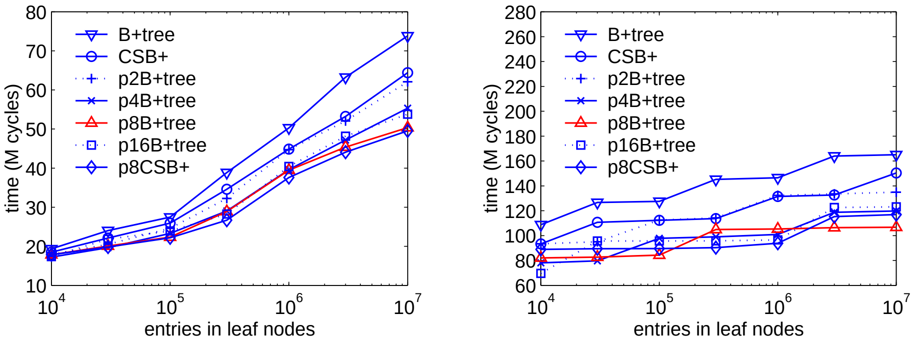
We first evaluate index search performance for B$^+$-Trees, CSB$^+$-Trees, p^wB$^+$-Trees (where $w = 2$, 4, 8, and 16), and p^8CSB$^+$-Trees (which combine our prefetching approach with CSB$^+$-Trees).

Varying the number of leaf nodes. Figure 7 shows the execution time of 100,000 random searches after bulkloading 10K, 30K, 100K, 300K, 1M, 3M, and 10M keys into the trees (nodes are 100% full except the root).[8] In the experiments shown in Figure 7(a), search operations are performed one immediately after another (the "warm cache" case); whereas in the experiments shown in Figure 7(b), the cache is cleared between each search (the "cold cache" case). Depending on the operations performed between the searches, the real-world performance of an index search would lie in between the two extremes: closer to the warm cache case for index joins, while often closer to the cold cache case for single value selections. From these experiments, we see that:(i) the B$^+$-Tree has the worst performance; (ii) the trees with wider nodes and prefetching support (pB$^+$-Trees, pCSB$^+$-Tree) all perform better than their non-prefetching counterparts (B$^+$-Tree, CSB$^+$-Tree); and (iii) the p^8B$^+$-Tree is comparable to or better than all other pB$^+$-Trees over the entire range of tree sizes. For warm caches, the speedup of the p^8B$^+$-Tree over the B$^+$-Tree is between a factor of 1.27 to 1.47. The warm cache speedup of the p^8B$^+$-Tree over the CSB$^+$-Tree is between a factor of 1.14 to 1.28 once the tree no longer fits in the L2 cache. Likewise, the cold cache speedups are 1.32 to 1.55 and 1.14 to 1.34, respectively.

The cold cache curves provide insight into the index search performance. The trend of every single curve is clearly shown in the cold cache experiment: the curves all increase in discrete large steps, and within the same step they increase only slightly. The large steps for a curve occur when the number of levels in the tree increases. This can be verified by examining Table 3, which depicts the number of levels in the tree for each data point plotted in Figure 7. Within a step, additional leaf nodes result in more keys in the root node (the other nodes in the tree remain full), which in turn increases the cost to search the root. The step-up trend is blurred in the warm cache curves because the top levels of the tree may remain in the cache. For different curves, we can see that generally the higher the tree structure, the larger the search cost; when trees are of the same height, the smaller node size yields better performance. We conclude that the performance gains for wider nodes stem mainly from the resulting decrease in tree height.

[7] Considering non-leaf nodes with **next-sibling** pointers, as is often done for concurrency control purposes [24], does not substantively alter our results.

[8] Note that when we refer to a number of keys, "K" and "M" correspond to 1000 and 1,000,000, respectively. When we refer to the size of a memory structure (e.g., a cache), however, "K" and "M" correspond to 1024 and 1,048,576, respectively.



(a) Warm cache (b) Cold cache

Figure 7: 100K searches after bulkloading 10K to 10M keys.

Table 3: The number of levels in trees for Figure 7.

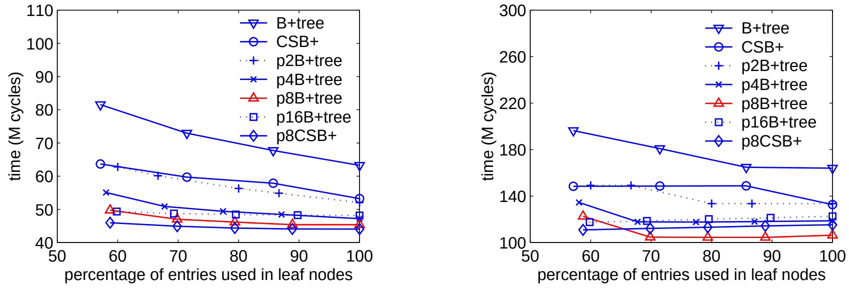
Tree Type	Number of Keys						
	10K	30K	100K	300K	1M	3M	10M
B^+-Tree	5	6	6	7	7	8	8
CSB^+-Tree	4	5	5	5	6	6	7
p^2B^+-Tree	4	4	5	5	6	6	6
p^4B^+-Tree	3	3	4	4	4	5	5
p^8B^+-Tree	3	3	3	4	4	4	4
$p^{16}B^+$-Tree	2	3	3	3	3	4	4
$pCSB^+$-Tree	3	3	3	3	3	4	4

There is a special case though. When the number of levels are the same, the p^2B^+-Tree and the CSB^+-Tree have very similar performance. This is because the second cache line in a p^2B^+-Tree node stores pointers, and the cost of retrieving these second lines is partly hidden by the key comparisons. By eliminating all but one pointer, the CSB^+-Tree has almost the same number of keys as the p^2B^+-Tree, resulting in similar key comparison costs.

Varying the bulkload factor. Figure 8 shows the effect on search performance of varying the bulkload factor. All the trees are bulkloaded with 3 million ⟨`key`, `tupleID`⟩ pairs, with bulkload factors 60%, 70%, 80%, 90%, and 100%. Because the actual number of used entries in leaf nodes in an experiment is the product of the bulkload factor and the maximum number of slots (rounded to the nearest integer), we computed and used the true percentage of used entries when plotting the data. Thus the plotted points may not be aligned with the target bulkload factors. As in the previous experiments, Figure 8 shows that: (i) the B^+-Tree has the worst performance; (ii) the trees with wider nodes and prefetching support (pB^+-Trees, $pCSB^+$-Tree) all perform better than their non-prefetching counterparts (B^+-Tree, CSB^+-Tree); and (iii) the p^8B^+-Tree is the best of all the pB^+-Trees.

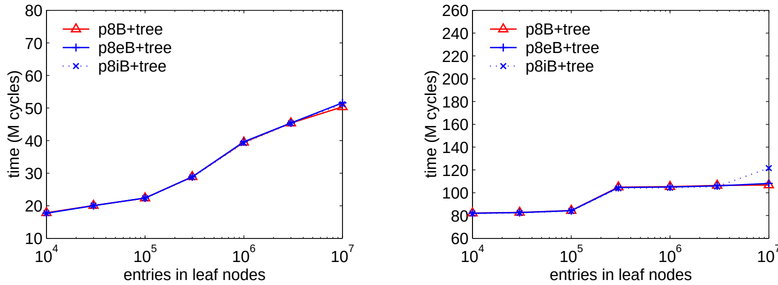
In the cold cache experiment, we see a step-down pattern in the curves: the steps correspond to the number of levels in the trees, for the tree height decreases (in a step-wise fashion) as the bulkload factor increases. Within a step, however, the curves increase slightly. This is because in our bulkload algorithms, the bulkload factor also determines the number of keys in non-leaf nodes. So the larger the bulkload factor, the larger the number of keys in each non-root node, and hence the larger the key comparison cost.

Searches on trees with range scan prefetching structures. Our next experiment determines whether the different structures for speeding up range scans have an impact on search performance. We use node



(a) warm cache (b) cold cache

Figure 8: 100K searches after bulkloading 3M keys with different bulkload factors.



(a) Warm cache (b) Cold cache

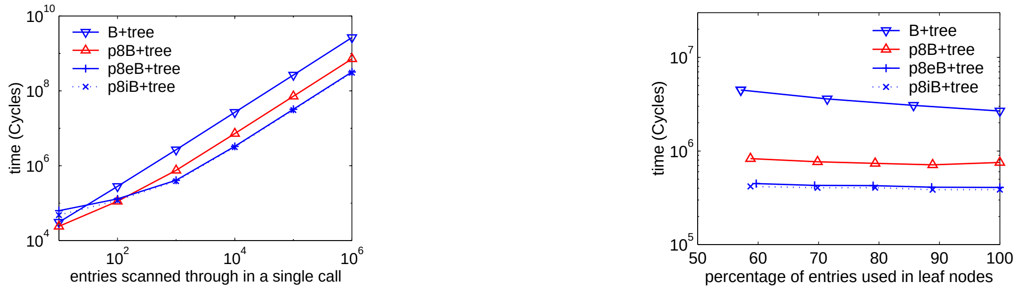
Figure 9: 100K searches after bulkloading 10K to 10M keys into p^{8}B$^+$-Trees with and without range scan prefetching structures.

width $w = 8$ for these experiments, because the p^{8}B$^+$-Tree resulted in the best search performance among the pB$^+$-Trees. Figure 9 compares the search performance of the p^{8}B$^+$-Tree, the p$_e^8$B$^+$-Tree, and the p$_i^8$B$^+$-Tree. The same experiments as in Figure 7 were performed. Recall that both the p$_e^8$B$^+$-Tree and the p$_i^8$B$^+$-Tree consume space in the tree structures relative to the p^{8}B$^+$-Tree: the maximum number of keys in leaf nodes is one fewer for the p$_e^8$B$^+$-Tree, and the maximum number of keys in bottom non-leaf nodes is one fewer for the p$_i^8$B$^+$-Tree. Figures 9(a) and 9(b) show that these differences have a negligible impact on search performance. In one cold cache case, when 10M keys are in the tree, the p$_e^8$B$^+$-Tree suffers from having one more level than the other two trees, but otherwise both the warm and cold cache performances are basically the same for all three trees, over the entire range of 10K to 10M keys.

4.3 Range Scan Performance

In our next set of experiments, we evaluate the effectiveness of our techniques for improving range scan performance. We compare B$^+$-Trees, p^{8}B$^+$-Trees, p$_e^8$B$^+$-Trees, and p$_i^8$B$^+$-Trees. As indicated above, we restrict our attention to node width $w = 8$ because this is the best width for searches, which are presumed to occur more frequently than range scans. As discussed in Section 4.1.2, we set the prefetching distance to 3 and the chunk size to 8 cache lines.

Varying the range size and the bulkload factor. Figure 10 shows the execution time of range scans while varying (a) the number of `tupleID`s to scan per request (i.e., the size of the range), or (b) the bulkload factor. Because of the large performance gains for pB$^+$-Trees, the execution time is shown on a logarithmic scale. In Figure 10(a), the trees are bulkloaded with 3 million ⟨`key`, `tupleID`⟩ pairs, using a 100% bulkload factor. Then 100 random starting `keys` are selected, and a range scan is requested for m `tupleID`s starting



(a) Varying the number of `tupleID`s scanned for each request

(b) Varying the bulkload factor

Figure 10: Range scan requests.

at that starting key value, for m = 10, 100, 1K, 10K, 100K, and 1M. The execution time plotted for each m is the average of the 100 starting `key`s. In Figure 10(b), the trees are bulkloaded with 3 million ⟨`key`, `tupleID`⟩ pairs, with bulkload factor 60%, 70%, 80%, 90%, and 100%. Then 100 random starting `key`s are selected, and a range scan is requested for 1000 tuple IDs starting at that value. Between the range scan requests, the caches are cleared, in order to more accurately reflect scenarios in which range scan requests are interleaved with other database operations or application programs (which would tend to evict any cache-resident nodes).

From the figures, we see that the $p_e^8B^+$-Tree and the $p_i^8B^+$-Tree achieve significantly better performance than the standard B^+-Tree, with a factor of **6.5 to 8.7 speedup** over the B^+-Tree, when the number of `tupleID`s scanned ranges from 1K to 1M. In Figure 10(b), we see that as the bulkload factor decreases, the number of leaf nodes to scan increases (we must scan past an increasing number of empty slots), and our prefetching schemes achieve even larger speedup. The plots reveal the contribution to the speedup of two of the pB^+-Tree techniques. First, extending the node size and then prefetching all the cache lines in a node while (searching and) scanning, results in a speedup of 3.5 to 3.7 (this is revealed in the speedup of the p^8B^+-Tree over the B^+-Tree). Second, also prefetching one or more leaf nodes ahead results in an additional speedup of around 2 (this is revealed in the speedup of both the $p_e^8B^+$-Tree and the $p_i^8B^+$-Tree over the p^8B^+-Tree). Note that the performance of the $p_e^8B^+$-Tree and the $p_i^8B^+$-Tree are nearly identical, indicating that there is no compelling need to build an external prefetching structure.[9]

When the number of `tupleID`s scanned is much smaller than 1K, however, the cost of the start-up phase in our prefetching schemes shows up in the performance curves. When scanning only 100 `tupleID`s, pB^+-Trees are only twice as fast as standard B^+-Trees. Moreover, when scanning only 10 `tupleID`s, both the $p_e^8B^+$-Tree and the $p_i^8B^+$-Tree are slower than the B^+-Tree, while the p^8B^+-Tree is only slightly faster than the B^+-Tree. This suggests a scheme in which the prefetching of the $p_e^8B^+$-Tree or the $p_i^8B^+$-Tree is employed only if the range is expected to be greater than 100 `tupleID`s. Estimating the size of the range can be done either (i) by a query optimizer using standard techniques such as histograms, or (ii) by simultaneously searching for both the starting and ending leaves of the range and then seeing how far apart they are.

Large segmented range scans. We next consider large range scans. In practice, a large range scan may be broken up into smaller segments, to permit other operations and queries to proceed or to avoid overflowing the return buffer. For example, an indexed scan providing sorted input to a sort-merge join operator will have its return buffer consumed at a rate dependent on the data profile of the operator's other input. Figure 11 shows the execution time for performing segmented range scans: each scan consists of a search for the starting key, followed by 1000 range scan requests, each scanning (and placing into the return buffer) the next segment of 1000 ⟨`key`, `tupleID`⟩ pairs, for a total of 1M pairs. The trees are bulkloaded with 3 million ⟨`key`, `tupleID`⟩ pairs, with bulkload factor 60%, 70%, 80%, 90%, and 100%. The reported execution times are the average of 100 segmented range scans, starting from 100 randomly selected starting

[9] This conclusion is dependent on the ratio of w to k. In other scenarios with different ratios, such as when prefetching to hide both memory and disk latencies, the flexibility of the external structure may be needed.

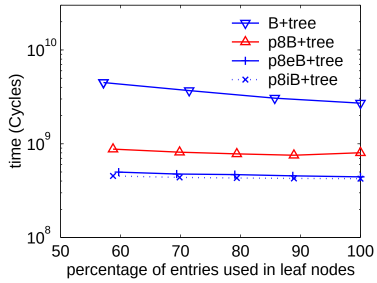


Figure 11: Large (segmented) range scans with varying bulkload factors.

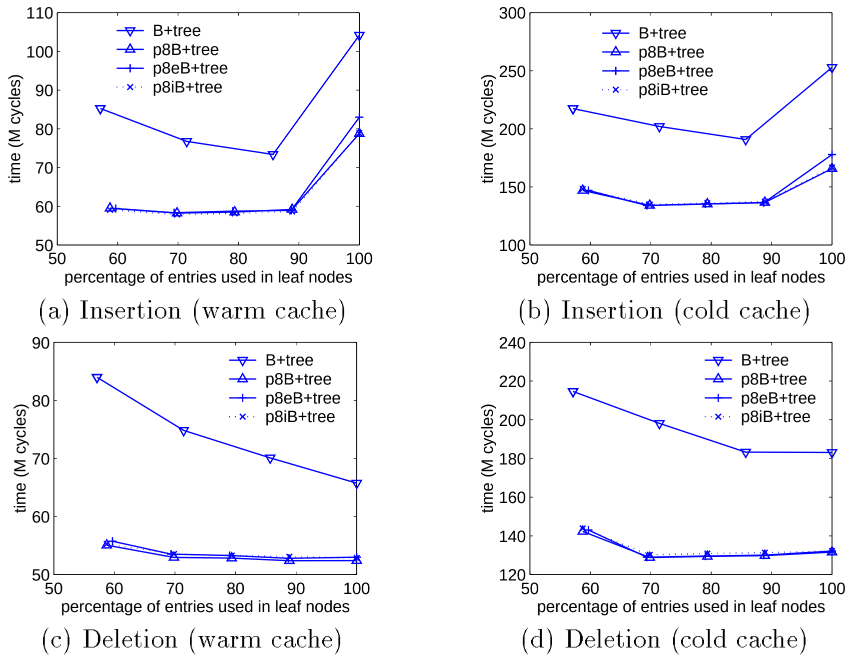


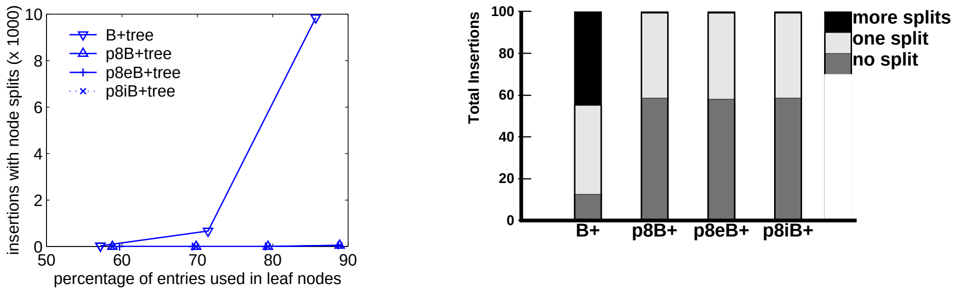
Figure 12: 100K update operations with varying bulkload factors.

keys. The performance gains for segmented range scans (Figure 11) are similar to those for non-segmented range scans (Figure 10).

4.4 Update Performance

One of our goals is to achieve good update performance as well as improving search and range scan. Our next set of experiments compare the update performance for B^+-Trees, p^8B^+-Trees, $p_e^8B^+$-Trees, and $p_i^8B^+$-Trees. Figure 12 depicts the execution time for 100K random insertions or deletions on a tree bulkloaded with 3 million ⟨**key**, **tupleID**⟩ pairs, with bulkload factor 60%, 70%, 80%, 90%, and 100%. We find that all three pB^+-Tree schemes perform roughly the same, and all are significantly faster than the B^+-Tree. For example, when the bulkload factor is 100%, the pB^+-Trees achieve at least a 1.24 speedup over the B^+-Tree in the warm cache case, and at least a 1.38 speedup in the cold cache case, for both insertions and deletions. This is somewhat surprising, given the additional overheads for maintaining the $p_e^8B^+$-Tree jump-pointer arrays.

There are two primary factors contributing to the faster update times for pB^+-Trees compared with the B^+-Tree. First, search is an integral part of both insertion and deletion, and search has been improved by



(a) Trees with empty slots (b) Trees that are 100% full

Figure 13: Insertion analysis.

our wider node prefetching scheme. Second, node splits are fewer for wider nodes. As shown in Figure 13(a), when the bulkload factor is 60% to 90%, the number of insertions that cause node splits is very low for all the pB$^+$-Trees. Thus the major cost of insertion is search. Even for the B$^+$-Tree, fewer than 10% of the insertions cause node splits. In this range, the curves in Figures 12(a) and (b) show similar trends to those in Figure 8. Thus we conclude that updates times for the pB$^+$-Tree are smaller than for the B$^+$-Tree due to its faster search times. On the other hand, when the trees are full, many insertions will cause node splits, as shown in Figure 13(b). But B$^+$-Trees suffer far more node splits, due to their smaller nodes, and over 40% of the insertions result in a more costly *non-leaf node* split. Wider nodes reduce the number of insertions resulting in node splits, especially those resulting in expensive non-leaf node splits. Though the cost of each pB$^+$-Tree node split is greater, this cost is more than compensated by the combined effect of faster search and fewer node splits.

Because both pB$^+$-Trees and B$^+$-Trees use lazy deletion, very few deletions result in a deleted node or a key redistribution, across the range of bulkload factors. Thus, the performance gains for pB$^+$-Tree deletions are due solely to the faster search.

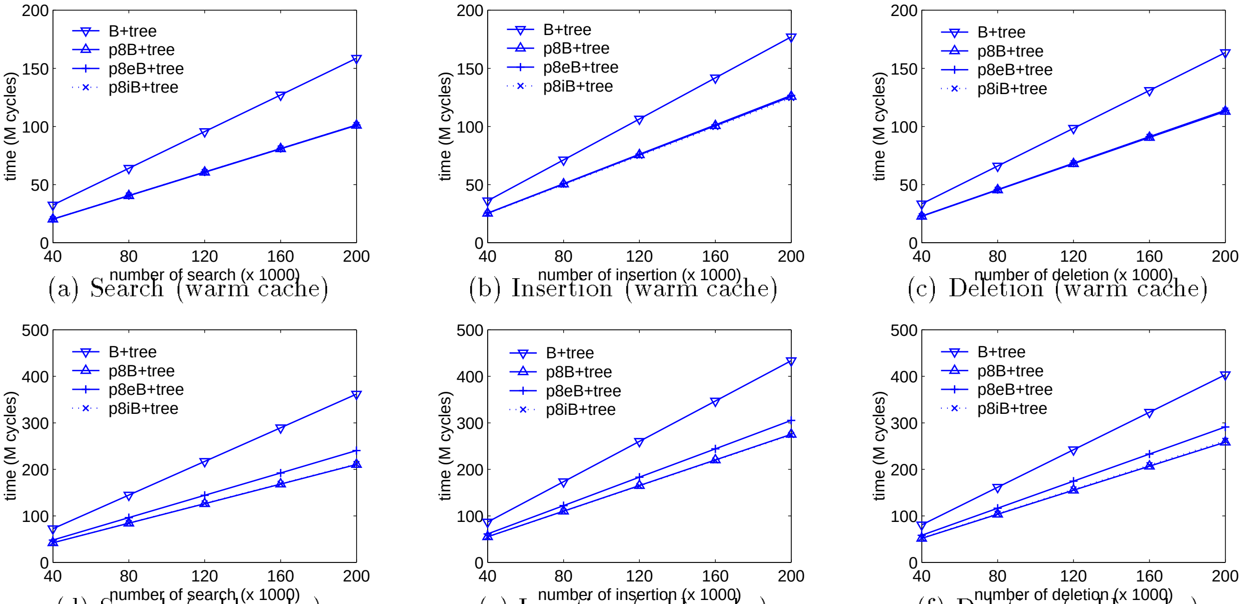
4.5 Operations on Mature Trees

Our next set of experiments show the performance of index operations on *mature trees* [21]. To obtain a mature tree, we use the same method as Rao and Ross [21]: we first bulkload a tree with 0.4 million ⟨`key`, `tupleID`⟩ pairs and then insert 3.6 million ⟨`key`, `tupleID`⟩ pairs. We compare the B$^+$-Tree, the p^{8}B$^+$-Tree, the p$_e^8$B$^+$-Tree, and the p$_i^8$B$^+$-Tree. Figure 14 shows the execution time for performing up to 200K random searches, insertions, or deletions. Figure 15 shows the execution times for: (a) range scans while varying the number of `tupleID`s; and (b) large (segmented) range scans. We find similar performance for mature trees as in previous experiments for trees immediately after bulkloads.

We can now compare the insertion performance of pB$^+$-Trees versus the CSB$^+$-Tree. In [21], the same experiments on mature trees showed that the CSB$^+$-Tree could be 25% worse than the B$^+$-Tree in insertion performance. This is because the CSB$^+$-Tree requires the sibling nodes to be moved when a node splits. The pB$^+$-Trees achieve better insertion performance than the B$^+$-Tree, which is better than the CSB$^+$-Tree. Thus for modern memory systems, all three pB$^+$-Trees are significantly faster than the CSB$^+$-Tree, for all the main operations of an index structure.

4.6 Sensitivity Analysis

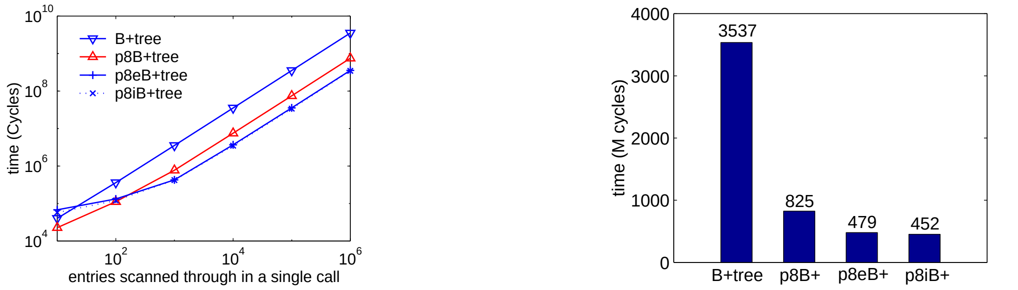
In our next set of experiments, we study the sensitivity of the performance gains to variations in (a) the memory system's normalized bandwidth B, (b) the prefetching distance k used, and (c) the chunk size c used. we will study the sensitivity of search performance to B and the sensitivity of scan performance to k and c. (k and c, which are parameters used in prefetching scan algorithm, do not affect search performance, and increasing memory bandwidth will proportionally increase prefetching scan performance as the major



(a) Search (warm cache) (b) Insertion (warm cache) (c) Deletion (warm cache)

(d) Search (cold cache) (e) Insertion (cold cache) (f) Deletion (cold cache)

Figure 14: 200K operations on mature trees.



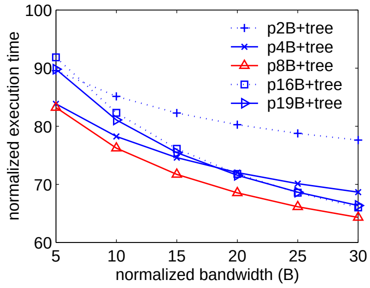
(a) Varying the number of `tupleID`s scanned for each request (b) Large (segmented) range scans

Figure 15: Range scan performance on mature trees.

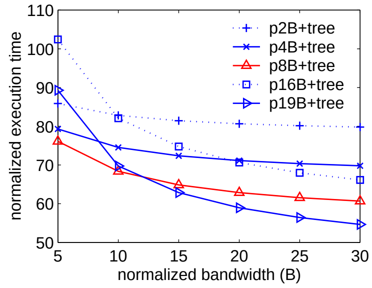
cost of prefetching scan is expected to be accessing memory of leaf nodes in a pipelined fashion) Recall that in our previous experiments, $B = 15$, $k = 3$, and $c = 8$.

Varying the normalized bandwidth. Figures 16(a) and 16(b) compare the search performance of the pB$^+$-Tree to the B$^+$-Tree, when the normalized bandwidth B ranges from 5 to 30. Because the optimal node width of a pB$^+$-Tree depends on B (recall equation (2) of Section 2), we show the search performance for widths w ranging from 2 to 19. Note that the performance of a B$^+$-Tree is independent of B, since it cannot exploit additional memory hierarchy bandwidth. The three variants of the pB$^+$-Tree have nearly identical search performance (recall Figure 9), so we plot only the p^{w}B$^+$-Tree performance.

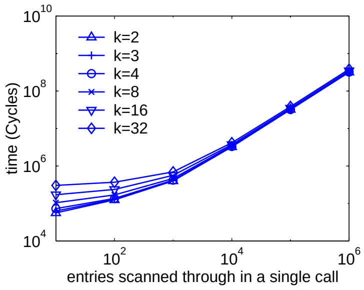
In this experiment, 3 million ⟨`key`, `tupleID`⟩ pairs are bulkloaded into each tree, followed by 100K random searches. Figures 16(a) and 16(b) show the execution time for the p^{w}B$^+$-Tree searches normalized to the execution time for the B$^+$-Tree searches. For each curve, we observe better relative performance with larger B, as the pB$^+$-Trees are able to exploit this additional bandwidth. Even in the somewhat pessimistic case where $B = 5$, the p^{8}B$^+$-Tree still achieves significant speedups: 1.2 and 1.3 for warm and cold caches, respectively. The cold cache performance corresponds to equation (2) in Section 2 with $N = 3$ million and



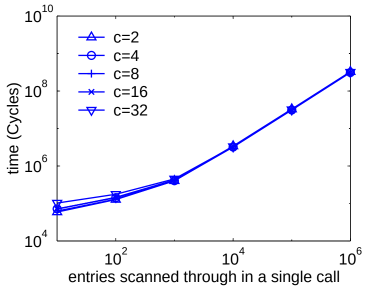
(a) Search varying memory bandwidth (warm cache)



(b) Search varying memory bandwidth (cold cache)



(c) Scan varying prefetching distance (p^8_eB$^+$-Tree)



(d) Scan varying chunk size (p^8_eB$^+$-Tree)

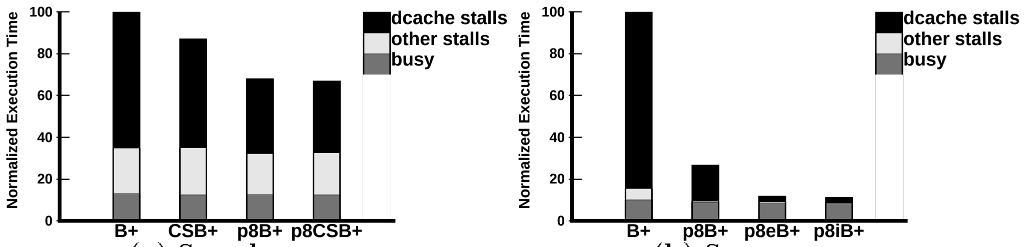
Figure 16: Sensitivity analysis

$m = 8$. As shown in Figure 16(b), the optimal value for w increases when B gets larger: p^{8}B$^+$-Tree is the best when B is small, while p^{19}B$^+$-Tree outperforms all others when $B \geq 15$.[10] For this particular number of keys (i.e. 3 million), the number of levels in the p^{2}B$^+$-Tree, p^{4}B$^+$-Tree, p^{8}B$^+$-Tree, p^{16}B$^+$-Tree, and p^{19}B$^+$-Tree are 6, 5, 4, 4, and 3, respectively. As we observed earlier in equation (2), the relative cost of using prefetching to create a wider node decreases as B becomes larger, and therefore the optimal w also increases. However, for a given number of keys, the performance only improves with a larger node size of this actually decreases the number of levels in the tree. When the trees are of the same heights, as in the cases of p^{8}B$^+$-Tree and p^{16}B$^+$-Tree, larger nodes lead to larger cost.

Varying the prefetching distance and the chunk size. Figures 16(c) and 16(d) show the effect on scan performance when the prefetching distance k varies from 2 to 32, and when the chunk size c varies from 2 to 32, respectively. We perform the same experiments as described earlier (in Section 4.3) for Figure 10(a). A larger prefetching distance leads to more overhead when we overshoot the end of the range. When only a small number of `tupleID`s are scanned in one request (e.g., 10 `tupleID`s), the performance improves with smaller k. On the other hand, when a large number of `tupleID`s are scanned, the overshooting overhead—which is independent of the number scanned—has a minimal incremental effect on the overall scan performance. Recall as well that overshooting can be reduced given more accurate estimates of range sizes, as discussed in Section 4.3. We conclude that the performance is not particularly sensitive to moderate increases in the prefetching distance.

As we see in Figure 16(d), varying the chunk size has only a minimal effect on scan performance, because the prefetching distance is still quite small compared with the number of leaf pointers in a chunk. When only a small number of `tupleID`s are scanned in one request, prefetching a larger chunk results in some degradation of performance, although it is quite small.

[10] Note that as shown in Section 2, $w = 8$ is the optimal value for $B = 15$ computed by averaging over N from 10^3 to 10^9. But this is not necessarily the optimum for a particular N—e.g., $N = 3$ million in Figure 16(b).



(a) Search (b) Scan

Figure 17: Impact of various forms of pB$^+$-Trees on the cache performance of index search and range scan.

4.7 Cache Performance

Finally, our last set of experiments present a more detailed cache performance study, using two representative experiments: one for index search and one for index range scan. A central claim of this paper is that the demonstrated speedups for pB$^+$-Trees are obtained by effectively limiting the exposed miss latency of previous approaches. In these experiments, we confirm that claim.

Our starting point is the experiments presented earlier in Figure 1 which illustrated the poor cache performance of existing B$^+$-Trees on index search and scan. We reproduce those results now in Figure 17, along with several variations of our pB$^+$-Trees. Figure 17(a) corresponds to the experiment shown earlier in Figure 7(a) with 10 million ⟨`key`, `tupleID`⟩ pairs, and Figure 17(b) corresponds to the experiment shown earlier in Figure 10(a) with 1 million `tupleID`s scanned.

Each bar in Figure 17 represents execution time normalized to a B$^+$-Tree, and is broken down into the following three categories that explain what happened during all potential *graduation slots*.[11] 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. Note that the *dcache stalls* section is only a first-order approximation of the performance loss due to data cache misses: these delays also exacerbate subsequent data dependence stalls, thereby increasing the number of *other stalls*.

As we see in Figure 17, pB$^+$-Trees significantly reduce the amount of exposed miss latency (i.e. the *dcache stalls* component of each bar). For the index search experiments, we see that while CSB$^+$-Trees eliminated 20% of the data cache stall time that existed with B$^+$-Trees, p^{8}B$^+$-Trees eliminate *45%* of this stall time, thus resulting in an overall speedup of *1.47* (compared with 1.15 for CSB$^+$-Trees). A significant amount of data cache stall time still remains for index searches, since we still experience the full miss latency each time we move down a level in the tree (unless the node is already in the cache due to previous operations). Eliminating this remaining latency appears to be difficult, as we will discuss in the next section. In contrast, we achieve nearly ideal performance for the index range scan experiments shown in Figure 17, where both p_e^8B$^+$-Trees and p_i^8B$^+$-Trees eliminate *97%* of the original data cache stall time, resulting in an impressive *eightfold overall speedup*. These results demonstrate that the pB$^+$-Tree speedups are indeed primarily due to a significant reduction in the exposed miss latency.

5 Discussion

We now discuss several possible improvements to pB$^+$-Trees and related issues. While our approach of using prefetching to create wider nodes improves search performance by a factor of 1.2–1.5, we still suffer a full cache miss latency at each level of the tree. Unfortunately, this is a very difficult problem to solve given: (i) the data dependence through the child pointer; (ii) the relatively large fanout of the tree nodes; and (iii)

[11] 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.

the fact that it is equally likely that any child will be visited (assuming uniformly distributed random search keys). While one might consider prefetching the children or even the grandchildren of a node in parallel with accessing the node, there is a duality between this and simply creating wider nodes. Compared with our approach, prefetching children or grandchildren suffers from (a) additional storage overhead for the children and grandchildren pointers, and (b) the restriction that the "size" of a node (i.e., the number of cache lines prefetched) can only grow by multiples of the tree fanout.

Although we have described our range scan algorithm for the case when the `tupleID`s are copied into a return buffer, other variations are only slightly more complex. For example, returning tuples instead of `tupleID`s involves only the additional step of prefetching the tuple once the `tupleID` has been identified. Moreover, if the index stores `tupleID`s with duplicate keys in separate lists, our prefetching approach could be used to retrieve the addresses to the `tupleID` lists, then the `tupleID`s, and finally the tuples themselves.

Extending the idea of adding pointers to the bottom non-leaf nodes, it is possible to use *no additional pointers at all*. Potentially, we could retain all the pointers from the root to the leaf during the search, and then keep moving this set of pointers, sweeping through the entire range prefetching the leaf nodes. Note that with wider nodes, trees are shallower and this scheme may be feasible.

Lehman and Carey, in an early paper on index structures for main memory databases, proposed and studied the T-Tree [11, 12]. At the time of their study (the mid-80's), the T-Tree outperformed the B^+-Tree, and was considered the index structure of choice for main memory databases for over a decade. However, more recent studies have shown that the B^+-Tree outperforms the T-Tree on modern processors [20], due in large part to the exponential growth these past 15 years in cache miss latency relative to processor speed.

Previous work has also considered key compression schemes (e.g., [4, 6, 8, 9, 17, 22]), in order to pack more keys into an index node. As with CSB^+-Trees, these techniques can be used in conjunction with our approach, as desired.

Although our discussions and experiments have focused on main memory databases, pB^+-Trees can also be used to improve both the I/O performance and the memory performance of disk-resident databases. Because the index node size for a disk-resident database is typically a disk page of 4KB, the fanout is much larger than with main memory indices. This may effect the benefits of using even wider nodes for searches. However, our range scan prefetching techniques applied to pages would likely continue to have a significant benefit. Furthermore, main memory performance is important even for disk-resident databases, so it would be interesting to apply our methods for both cache lines and pages, and quantify the overall performance gains.

6 Conclusions

While eliminating child pointers through data layout techniques has been shown to significantly improve main memory B^+-Tree search performance, a large fraction of the execution time for a search is still spent in data cache stalls, and index insertion performance is hurt by these techniques. Moreover, the cache performance of index scan (another important B^+-Tree operation) has not been studied. In this paper, we explored how prefetching could be used to improve the cache performance of index search, update, and scan operations. We proposed the *Prefetching B^+-Tree* (pB^+-Tree) and evaluated its effectiveness in modern memory systems.

We showed that the optimal B^+-Tree node size is often wider than a cache line on a modern machine, when prefetching is used to retrieve the pieces of a node, effectively overlapping multiple cache misses. Our results can be summarized as follows:

- For index search, this prefetching technique achieves a speedup of 1.27 to 1.55 over the B^+-Tree, by decreasing the height of the tree.
- For index updates (insertions and deletions), the technique achieves a speedup of 1.24 to 1.52 over the B^+-Tree, due to the faster search and the less frequent node splits with wider nodes.
- For index scan, the technique achieves a speedup of 3.5 to 3.7 over the B^+-Tree, again due to the faster search and wider nodes. Moreover, we proposed *jump-pointer arrays*, which enable effective range scan prefetching across node boundaries. Overall, the pB^+-Tree achieves a speedup of **6.5 to 8.7** over the

B^+-Tree for range scans. We proposed two alternative implementations of jump-pointer arrays, with comparable performance.

From our results, we conclude that the cache performance of B^+-Tree indices can be greatly improved by exploiting the prefetching capabilities of state-of-the-art computer systems. We believe that this work makes an important contribution towards applying prefetching techniques to advantage throughout a DBMS.

7 Acknowledgments

We thank Berni Schiefer at IBM for his many helpful comments regarding this work.

References

[1] 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 International Conference on Very Large Data Bases (VLDB)*, pages 266–277, September 1999.

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

[3] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 3–14, January 2000.

[4] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.

[5] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, November 2000.

[6] P. Bohannon, P. McIlroy, and R. Rastogi. Improving Main-Memory Index Performance with Partial Key Information. Technical report, Bell Laboratories, November 2000.

[7] Z. Cvetanovic and R. E. Kessler. Performance Analysis of the Alpha 21264-Based Compaq ES40 System. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 192–202, June 2000.

[8] D. Ferguson. Bit-tree: A Data Structure for Fast File Processing. *Communications of the ACM*, 35(6):114–120, June 1992.

[9] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 370–379, February 1998.

[10] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, pages 15–26, June 1998.

[11] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 294–303, August 1986.

[12] T. J. Lehman and M. J. Carey. Query Processing in Main Memory Database Management Systems. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 239–250, May 1986.

[13] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 222–233, October 1996.

[14] C.-K. Luk and T. C. Mowry. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Transactions on Computers*, 48(2):134–141, February 1999.

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

[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 62–73, October 1992.

[17] W. K. Ng and C. V. Ravishankar. Block-oriented Compression Techniques for Large Statistical Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):314–328, March/April 1997.

[18] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 233–242, May 1994.

[19] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 307–318, October 1998.

[20] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 78–89, September 1999.

[21] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of ACM SIGMOD 2000 International Conference on Management of Data (SIGMOD)*, pages 475–486, May 2000.

[22] M. Ronström. *Design and Modelling of a Parallel Data Server for Telecom Applications.* PhD thesis, Linköping University, April 1998.

[23] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 510–521, September 1994.

[24] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw Hill, New York, New York, 3rd edition, 1997.

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