Making B^+-Trees Cache Conscious in Main Memory

Jun Rao
Columbia University
junr@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Abstract

Previous research has shown that cache behavior is important for main memory index structures. Cache conscious index structures such as Cache Sensitive Search Trees (CSS-Trees) perform lookups much faster than binary search and T-Trees. However, CSS-Trees are designed for decision support workloads with relatively static data. Although B^+-Trees are more cache conscious than binary search and T-Trees, their utilization of a cache line is low since half of the space is used to store child pointers. Nevertheless, for applications that require incremental updates, traditional B^+-Trees perform well.

Our goal is to make B^+-Trees as cache conscious as CSS-Trees without increasing their update cost too much. We propose a new indexing technique called "Cache Sensitive B^+-Trees" (CSB^+-Trees). It is a variant of B^+-Trees that stores all the child nodes of any given node contiguously, and keeps only the address of the first child in each node. The rest of the children can be found by adding an offset to that address. Since only one child pointer is stored explicitly, the utilization of a cache line is high. CSB^+-Trees support incremental updates in a way similar to B^+-Trees.

We also introduce two variants of CSB^+-Trees. Segmented CSB^+-Trees divide the child nodes into segments. Nodes within the same segment are stored contiguously and only pointers to the beginning of each segment are stored explicitly in each node. Segmented CSB^+-Trees can reduce the copying cost when there is a split since only one segment needs to be moved. Full CSB^+-Trees preallocate space for the full node group and thus reduce the split cost. Our performance studies show that CSB^+-Trees are useful for a wide range of applications.

*This research was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by an NSF Young Investigator Award, by NSF grant number IIS-98-12014, and by NSF CISE award CDA-9625374.

1 Introduction

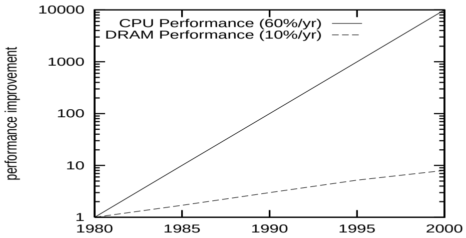


Figure 1: CPU-memory Performance Imbalance

As random access memory gets cheaper, it becomes increasingly affordable to build computers with large main memories. The recent "Asilomar Report" ([BBC+98]) predicts: "Within ten years, it will be common to have a terabyte of main memory serving as a buffer pool for a hundred-terabyte database. All but the largest database tables will be resident in main memory." But main memory data processing is not as simple as increasing the buffer pool size. An important issue is cache behavior. The traditional assumption that memory references have uniform cost is no longer valid given the current speed gap between cache access and main memory access. [ADW99] studied the performance of several commercial database management systems in main memory. The conclusions they reached is that a significant portion of execution time is spent on second level data cache misses and first level instruction cache misses. Further more, CPU speeds have been increasing at a much faster rate (60% per year) than memory speeds (10% per year) as shown in Figure 1. So, improving cache behavior is going to be an imperative task in main memory data processing.

Index structures are important even in main memory database systems. Although there are no disk accesses, indexes can be used to reduce overall computation time without using too much extra space. Index structures are useful for single value selection, range queries and indexed nested loop joins. With a large amount of RAM, most of the indexes can be memory resident. In our earlier work [RR99], we studied the performance of main memory index structures and found that B^+-Trees are more cache conscious than binary search trees and T-Trees [LC86]. We proposed a new index structure called "Cache-Sensitive Search Trees" (CSS-Tree) that has even better cache behavior than a B^+-Tree. CSS-Trees augment binary search by storing a directory structure on top of the sorted list of elements. CSS-Trees avoid storing child pointers explicitly by embedding the directory structure in an array sequentially, and thus have a better utilization of each cache line. Although this approach improves the searching speed, it also makes incremental updates difficult since the relative positions between nodes are important. As a result, we have to batch updates and rebuild the CSS-Tree once in a while.

In this paper, we introduce a new index structure called the "Cache-Sensitive B^+-Tree" (CSB^+-Tree) that retains the good cache behavior of CSS-Trees while at the same time being able to support incremental updates. A CSB^+-Tree has a structure similar to a B^+-Tree. Instead of storing all the child pointers explicitly, a CSB^+-Tree puts all the child nodes for a given node contiguously in an array and stores only the pointer to the first child node. Other child nodes can be found by adding an offset to the first-child pointer. This approach allows good utilization of a cache line. Additionally, CSB^+-Trees can support incremental updates in a way similar to B^+-Trees.

CSB^+-Trees need to maintain the property that sibling nodes are contiguous, even in the face of updates. We call a set of sibling nodes a *node group*. There are several ways to keep node groups contiguous, all of which involve some amount of copying of nodes when there is a split. We present several variations on the CSB^+-Tree idea that differ in how they achieve the contiguity property. The simplest approach is to deallocate a node group and allocate a new larger node group on a split. "Segmented" CSB^+-Trees reduce the update cost by copying just segments of node groups. "Full" CSB^+-Trees pre-allocate extra space within node groups, allowing easier memory management and cheaper copying operations.

We compare the various CSB^+-Tree methods with B^+-Trees and CSS-Trees, both analytically and experimentally. We demonstrate that Full CSB^+-Trees dominate B^+-Trees in terms of both search and update times, while requiring slightly more space than B^+-Trees. Other CSB^+-Tree variants that take substantially less space than B^+-Trees also outperform B^+-Trees when the workload has more searches than updates.

It is now well accepted that many applications can benefit from having their data resident in a main memory database. Our results are significant for main memory database performance because index operations are frequent. Full CSB^+-Trees are the index structure of choice in terms of time performance for all workloads. For applications with workloads where there are more searches than updates, the other CSB^+-Tree variants also outperform B^+-Trees. Such applications include on-line shopping where the inventories are queried much more often than changed, and digital libraries, where the frequency of searching for an article is higher than that of adding an article.

The rest of this paper is organized as follows. In Section 2 we survey related work on cache optimization. In Section 3 we introduce our new CSB^+-Tree and its variants. In Section 4 we compare the different methods analytically. In Section 5 we present a detailed experimental comparison of the methods. We conclude in Section 6.

2 Related Work

2.1 Cache Memories and Cache Conscious Techniques

Cache memories are small, fast static RAM memories that improve program performance by holding recently referenced data [Smi82]. A cache can be parameterized by capacity, block (cache line) size and associativity, where capacity is the size of the cache, block size is the basic transferring unit between cache and main memory, associativity determines how many slots in the cache are potential destinations for a given address reference. Typical cache line sizes range from 32 bytes to 128 bytes.

Memory references satisfied by the cache, called hits, proceed at processor speed; those unsatisfied, called misses, incur a cache miss penalty and have to fetch the corresponding cache block from the main memory. Modern architectures typically have two levels of cache (L1 and L2) between the CPU and main memory. While the L1 cache can perform at CPU speed, the L2 cache and main memory accesses normally introduce latencies in the order of 10 and 100 cycles respectively. Cache memories can reduce the memory latency only when the requested data is found in the cache. This mainly depends on the memory access pattern of the application. Thus, unless special care is taken, memory latency will become an increasing performance bottleneck, preventing applications from fully exploiting the power of modern hardware.

Some previous work on cache conscious techniques were summarized in [RR99]. Recently, [BMK99] proposed to improve cache behavior by storing tables vertically and by using a more cache conscious join method.

2.2 Cache Optimization on Index Structures

B$^+$-Trees. We assume that the reader is familiar with the B$^+$-Tree index structure [Com79]. In [RR99] an analysis of the search time for B$^+$-Trees in a main-memory system was performed. The search times were not as good as CSS-Trees because at least half of each B$^+$-Tree node is taken up by pointers rather than keys. Compared with CSS-Trees, B$^+$-Trees utilize fewer keys per cache line, resulting in more cache accesses and more cache misses.

On the other hand, B$^+$-Trees have good incremental performance. Insertion and deletion are relatively efficient, and the requirement that nodes be half full bounds the size and depth of the tree.

In a main memory system, a cache line is the basic transferring unit (same as a page in a disk-based system). As observed in [RR99, CLH98], B$^+$-Trees with node size of a cache line have close to optimal performance.

CSS-Trees. CSS-Trees were proposed in [RR99]. They improve on B$^+$-Trees in terms of search performance because each node contains only keys, and no pointers. Child nodes are identified by performing arithmetical operations on array offsets. Compared with B$^+$-Trees, CSS-Trees utilize more keys per cache line, and thus need fewer cache accesses and fewer cache misses.

The use of arithmetic to identify children requires a rigid storage allocation policy. As argued in [RR99], this kind of policy is acceptable for static data updated in batches, typical of a decision-support database. However, there is no efficient means to update CSS-Trees incrementally; the whole index structure must be rebuilt.

Other Pointer Elimination Techniques. [TMJ98] proposed a Pointer-less Insertion Tree (PLI-Tree). A PLI-Tree is a variant of a B$^+$-Tree. It allocates nodes in a specific order so that child nodes can be found through arithmetic calculations. As a result, PLI-Trees don't have to store child pointers explicitly. However, PLI-Trees are designed for append-only relations, such as backlogs where data is inserted in transaction timestamp order. All insertions are done in the rightmost leaf only and node splitting never occurs.

In [Ker89], the author mapped a binary search tree to an array in an unconventional way, calling the resulting structure a Virtual Tree (V-Tree). V-Trees can use a simple search procedure that uses implicit search information rather than explicit search pointers. Although V-Trees were shown to have better search performance (when the paper was published), they impose an upper bound on the size of the indices. Also, the maintenance cost starts to deteriorate when the area set aside for holding the index is nearly full.

To summarize, pointer elimination is an important technique in cache optimization since it increases the utilization of a cache line. The effect of pointer elimination depends on the relative key size. Keys of size much larger than the pointer size may reduce the impact of pointer elimination. If such is the case, we can put all distinct key values in a *domain* and store in place only the IDs as described in [RR99]. Thus, we assume that typical keys have the same size as integers. In the near future, we are going to have 64-bit operating systems. This means each pointer will be 8 bytes, instead of 4 bytes. Potentially, pointers can take more space than data. So pointer elimination will be even more important in the future. However, removing pointers completely often introduces some restrictions. For example, PLI-Trees require data to be inserted in order and CSS-Trees and V-Trees don't support incremental updates very well. As we will see shortly, we use a *partial* pointer elimination technique in CSB$^+$-Trees. By doing this, we avoid introducing new restrictions while at the same time being able to optimize cache behavior.

Finally, we don't address concurrency control and recovery in this paper. We'd like to investigate the impact of these issues on main memory indexing in the future.

3 Cache Sensitive B$^+$-Trees

Our goal is to obtain cache performance close to that of CSS-Trees, while still enabling the efficient incremental updates of B^+-Trees. We achieve this goal by balancing the best features of the two index structures. Our tree structure, which we call a CSB^+-Tree, is similar to a B^+-Tree in the way it handles updates. However, a CSB^+-Tree has fewer pointers per node than a B^+-Tree. By having fewer pointers per node, we have more room for keys and hence better cache performance.

We get away with fewer pointers by using a limited amount of arithmetic on array offsets, together with the pointers, to identify child nodes. For simplicity of presentation, we initially present a version of CSB^+-Trees in which a node contains exactly one pointer. Sometimes we simply use the term CSB^+-Tree to refer to this version when the context is clear. In Section 3.2 we will describe variants with more pointers per node. The number of pointers per node is a parameter that can be tuned to obtain good performance under particular workloads. We describe another variant of CSB^+-Trees that further reduces split cost in Section 3.3.

3.1 Cache Sensitive B^+-Trees with One Child Pointer

A CSB^+-Tree is a balanced multi-way search tree. Every node in a CSB^+-Tree of order d contains m keys, where $d <= m <= 2d$. A CSB^+-Tree puts all the child nodes of any given node into a *node group*. Nodes within a node group are stored contiguously and can be accessed using an offset to the first node in the group.[1] Each internal node in a CSB^+-Tree has the following structure:

nKeys	:number of keys in the node
firstChild	:pointer to the first child node
keyList[2d]	:a list of keys.

Each leaf node stores a list of <key, tuple ID> pairs, the number of these pairs, and two sibling pointers.[2]

Since a CSB^+-Tree node needs to store just one child pointer explicitly, it can store more keys per node than a B^+-Tree. For example, if the node size (and cache line size) is 64 bytes and a key and a child pointer each occupies 4 bytes, then a B^+-Tree can only hold 7 keys per node whereas a CSB^+-Tree can have 14 keys per node. This gives CSB^+-Tree two kinds of benefit: (a) a cache line can satisfy (almost) one more level of comparisons and thus the number of cache lines needed for a search is fewer; (b) the fan out of each node is larger, which means it uses less space. Figure 2 shows a CSB^+-Tree of order 1. Each dashed box represents a node group. The arrows from the internal nodes represent the first child pointers. All the nodes within a node group are physically adjacent to each other. In this example, a node group can have no more than three nodes within it. Note that grouping is just a physical ordering property, and does not have any associated space overhead.

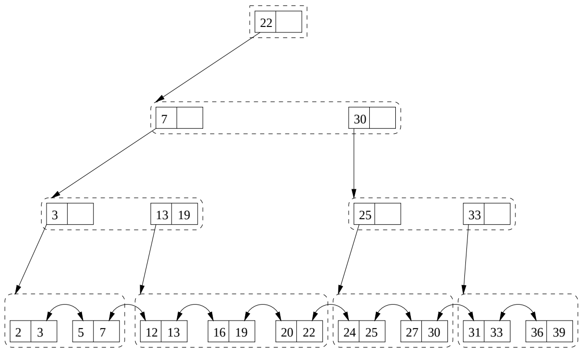


Figure 2: A CSB^+-Tree of Order 1

3.1.1 Operations on a CSB^+-Tree

In this section, we consider bulkload, search, insert and delete operations on CSB^+-Trees.

Bulkload. A typical bulkloading algorithm for B^+-Trees is to keep inserting sorted leaf entries into the rightmost path from the root. However, this method can be expensive if used for CSB^+-Trees since nodes in the same node group are not created sequentially. A more efficient bulkloading method for CSB^+-Trees is to build the index structure level by level. We allocate space for all the leaf entries. We then calculate how many nodes are needed in the higher level and then allocate a continuous chunk of space for all the nodes in this level. We then fill in the entries of nodes in the higher level by copying the largest value in each node in the lower level. We also set the first child pointer in each higher level node. We repeat the process until the higher level has only one node and this node is designated as the root. Since all the nodes in the same level are contiguous when they are created, we don't have to do any additional copying to form a node group.

Search. Searching a CSB^+-Tree is similar to searching a B^+-Tree. Once we have determined the rightmost key K in the node that is smaller than the search key, we simply add the offset of K to the first-child pointer to get the address of the child node. (For values less than or equal to the leftmost key, the offset is 0.) So, for example, if

[1] [O'N92] also considers grouping nodes together in a disk-based B^+-Tree to improve I/O performance.

[2] see Section 5 for further discussion of how leaf nodes can be implemented.

K was the third key in the node, we would find the child using a `C` statement: `child = first_child + 3,` where `child` and `first_child` are pointers to nodes. There are several ways to search efficiently within a node; we defer further discussion until Section 3.1.2.

Insertion. Insertion into a CSB$^+$-Tree is also similar to that of a B$^+$-Tree. A search on the key of the new entry is performed first. Once the corresponding leaf entry is located, we determine if there is enough room in the leaf node. If there is, we simply put the new key in the leaf node. Otherwise, we have to split the leaf node.

When a leaf is split, there are two cases depending on whether the parent node has space for a new key. Suppose the parent node p has enough space. Let f be the first-child pointer in p, and let g be the node-group pointed to by f. We create a new node group g' with one more node than g. All the nodes from g are copied into g', with the node in g that was split resulting in two nodes within g'. We then update the first child pointer f in p to point to g', and de-allocate g.

A more complicated case arises when the parent node p is full and itself has to split. Again, let f be the first-child pointer in p, and let g be the node-group pointed to by f. In this case, we have to create a new node group g' and redistribute the nodes in g evenly between g and g'. Half the keys of p are transferred to a new node p', whose first-child pointer is set to g'. To achieve this split of p into p and p', the node-group containing p must be copied as in the first case above, or, if that node group is also full, we need to recursively split the parent of p. The parent node will then repeat the same process.

When there is a split, CSB$^+$-Trees have to create a new node group whereas B$^+$-Trees only need to create a new node. Thus when there are many splits, maintaining a CSB$^+$-Tree could be more expensive (we'll talk about how to reduce this cost in Section 3.2). On the other hand, CSB$^+$-Trees have the benefit of being able to locate the right leaf node faster. Potentially, we can reserve more space in a node group to reduce copying. We shall elaborate on this idea in Section 3.3.

Deletion. Deletion can be handled in a way similar to insertion. In practice, people choose to implement the deletion "lazily" by simply locating the data entry and removing it, without adjusting the tree as needed to guarantee 50% occupancy [Ram97]. The justification for lazy deletion is that files typically grow rather than shrink.

3.1.2 Searching within a Node

The most commonly used piece of code within all operations on a CSB$^+$-Tree is searching within a node. (The same is true for B$^+$-Trees.) So it's important to make this part as efficient as possible. We describe several approaches here.

The first approach, which we call the *basic* approach, is to simply do a binary search using a conventional while loop.

We can do much better than this approach through code expansion: As observed in [RR99], code expansion can improve the performance by 20% to 45%. Thus, our second approach is to unfold the while loop into `if-then-else` statements assuming all the keys are used. If we pad all the unused keys (`keyList[nKeys..2d-1]`) in a node with the largest possible key (or the largest key in the subtree rooted at this node), we are guaranteed to find the right branch. This approach avoids arithmetic on counter variables that are needed in the basic approach.

There are many possible unfoldings that are not equivalent in terms of performance. For example, consider Figure 3 that represents an unfolding of the search for a node with up to 9 keys. The number in a node in Figure 3 represents the position of the key being used in an `if` test. If only 5 keys were actually present, we could traverse this tree with exactly 3 comparisons. On the other hand, an unfolding that put the deepest subtree at the left instead of the right would need 4 comparisons on some branches. We hard code the unfolded binary search tree in such a way that the deepest level nodes are filled from right to left. Since keys in earlier positions have shorter paths, this tree favors the cases when not all the key positions are filled. We call this second approach the *uniform* approach because we use a hard-coded traversal that is uniform no matter how many keys are actually present in a node.

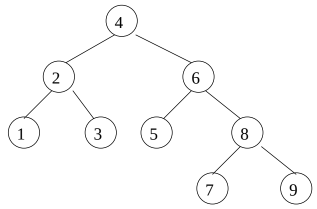


Figure 3: A Binary Search Tree with 9 Keys

The uniform approach could perform more comparisons than optimal. Consider Figure 3 again. If we knew we had only five valid keys, we could hard-code a tree that, on average, used 2.67 comparisons rather than 3. Our third approach is thus to hard-code all possible optimal search trees (ranging from

1 key to $2d$ keys). If we put all the hard-coded versions in an array of function pointers, we can call the correct version by indexing via the actual number of keys being used. Although this method avoids unnecessary comparisons, it introduces an extra function call, which can be expensive. Some `C` extensions (e.g., `gcc`) allow branching to a pointer variable that can be initialized via a program label. This trick allows us to inline the search code and jump to the beginning of the appropriate part and thus avoid the function call, paying just the cost of an extra array lookup and an extra jump at the end. This approach, however, increases the code size, which could be a problem when d is large. We call this approach the *variable* approach because the intra-node search method depends on the number of keys present.

3.2 Segmented Cache Sensitive B$^+$-Trees

Consider a cache-line of 128 bytes. Each node in a CSB$^+$-Tree can have a maximum of 30 keys. This means every node can have up to 31 children. A node group then has a maximum size of $31 * 128 \approx 4KB$. So every time a node split, we need to copy about 4KB of data to create a new node group. If the cache line were to get larger in future architectures, splitting a node would become more expensive.

One way to address this issue is to modify the node structure so that less copying takes place during a split. We divide the child nodes into segments and store in each node the pointers to each segment. Each segment forms a node group and only child nodes in the same segment are stored contiguously. For the sake of simplicity, we discuss only the two segment case in the rest of this section.

Our first thought is to fix the size of each segment. We start filling nodes in the first segment. Once the first segment is full, we begin to put nodes in the second segment. Now, if a new node falls in the second segment, we only need to copy nodes in the second segment to a new segment and we don't need to touch the first segment at all. However, if the new node falls in the first segment (and it's full), we have to move data from the first segment to the second one. Assuming random insertion, in the above example, the average data copied during a split will be reduced to $\frac{1}{2}(\frac{1}{2}+\frac{3}{4}) * 4KB = 2.5KB$.

Another approach is to allow each segment to have a different size. During the bulkload, we distribute the nodes evenly into the two segments. We also keep the size of each segment (actually, the size of the first segment is enough). Every time a new node is inserted, we only create a new segment for the segment the new node belongs to. We then update the size of the corresponding segment. In this approach, exactly one segment is touched on every insert (except when the parent also needs to split, in which case we have to copy both segments). If a new node is equally likely to fall into either segment, the amount of data to be copied on a split is $\frac{1}{2} * 4KB = 2KB$. As we can see, this approach can further reduce the cost of copying. In the rest of the paper, this approach is the segmented CSB$^+$-Tree we are referring to. A segmented CSB$^+$-Tree (order 2) with two segments is shown in Figure 4 (we put only 2 keys per leaf node though).

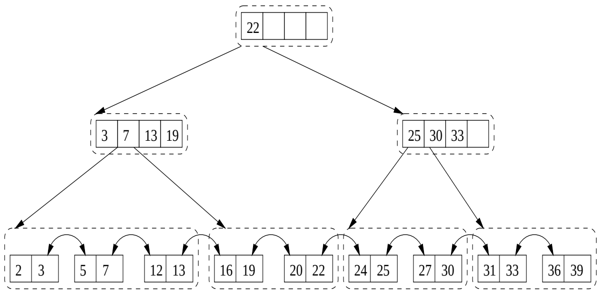


Figure 4: SCSB$^+$-Tree of Order 2 with 2 Segments

All tree operations can be supported for segmented CSB$^+$-Trees in a similar way to unsegmented CSB$^+$-Trees. However, finding the right child within each node is more expensive than the unsegmented case since now we have to find out which segment the child belongs to.

3.3 Full CSB$^+$-Trees

During a node split in a CSB$^+$-Tree, we deallocate a node group (say of size s) and allocate a node group of size $s + 1$. As a result, we pay some overhead for allocating and deallocating memory. If we were to *pre-allocate* space for a *full* node group whenever a node group is created, then we can avoid the bulk of the memory allocation calls. We need to allocate memory only when a node group (rather than a node) overflows. We call the variant of CSB$^+$-Trees that pre-allocates space for full node groups *full* CSB$^+$-Trees.

In full CSB$^+$-Trees, node splits may be cheaper than for CSB$^+$-Trees, even if one ignores the saving of the memory allocation overhead. In a CSB$^+$-Tree, when a node splits, we copy the full node group to a new one. In a full CSB$^+$-Tree, we can shift part (on average, half) of the node group along by one node, meaning we access just half the node group. Further, since the source and destination addresses for such a shift operation largely overlap,

Method	Branching Factor	Total Key Comparisons	Cache Misses	Extra Comparisons per Node
Full CSS-Trees	$m+1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m+1)}$	0
Level CSS-Trees	m	$\log_2 n$	$\frac{\log_2 n}{\log_2 m}$	0
B^+-Trees	$\frac{m}{2}$	$\log_2 n$	$\frac{\log_2 n}{\log_2 m-1}$	0
CSB^+-Trees	$m-1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-1)}$	0
CSB^+-Trees (t segments)	$m-2t+1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-2t+1)}$	$\log_2 t$
Full CSB^+-Trees	$m-1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-1)}$	0

Table 1: Search Time Analysis

the number of cache lines accessed is bounded by s. In modern architectures, a cache write miss often requires loading the corresponding cache line into the cache (a read miss) first before writing the actual data. On average, full CSB^+-Trees touch $0.5s$ nodes on a split, whereas CSB^+-Trees touch $2s$ (s reads and s writes). Perfectly balanced 2-segment CSB^+-Trees and 3-segment CSB^+-Trees will touch s and $0.67s$ nodes respectively.

Thus we would expect full CSB^+-Trees to outperform CSB^+-Trees on insertions. On the other hand, pre-allocation of space means that we are using additional space to get this effect. This is a classic space/time trade-off.

4 Time and Space Analysis

In this section, we analytically compare the time performance and the space requirement for different methods. In particular, we want to compare B^+-Trees, CSS-Trees and CSB^+-Trees. To simplify the presentation, we assume that a key, a child pointer and a tuple ID all take the same amount of space K. We let n denote the number of leaf nodes being indexed, c denote the size of a cache line in bytes, and t denote the number of segments in a segmented CSB^+-Tree. The number of slots per node is denoted by m, which can be derived using $m = \frac{c}{K}$. We assume each node size is the same as the cache line size. Those parameters and their typical values are summarized in Figure 5.

Parameter	Typical Value
K	4 bytes
n	10^7
c	64 bytes
t	2
$m = \frac{c}{K}$	16

Figure 5: Parameters and Their Typical Values

Table 1 shows the branching factor, total number of key comparisons, number of cache misses and number of additional comparisons of searching for each method. B^+-Trees have a smaller branching factor than CSS-Trees since they need to store child pointers explicitly. CSB^+-Trees have a branching factor close to CSS-Trees as fewer child pointers are stored explicitly. This leads to different number of cache misses for each of the methods. The larger the branching factor of a node, the smaller the number of cache misses. For each additional segment in CSB^+-Trees, the branching factor is reduced by 2 since we have to use one slot to store child pointers and another to store the size of the additional segment. Also, when there are multiple segments in CSB^+-Trees, we need to perform additional comparisons to determine which segment the child belongs to. The numbers for B^+-Trees and CSB^+-Trees assume that all the nodes are fully used. In practice, typically a B^+-Tree node is about 70% full [Yao78] and we have to adjust the branching factor accordingly.

Method	Accessed Cache Lines in a Split	Typical Values (cache lines)
B^+-Trees	2	2
CSB^+-Trees	$(m-1)*2$	30
CSB^+-Trees (t segments)	$\frac{(m-2t+1)*2}{t}$	13
Full CSB^+-Trees	$\frac{m-1}{2}$	7.5

Table 2: Split Cost Analysis

Table 2 shows the expected number of cache lines that need to be accessed during a split. Full CSB^+-Trees have a smaller number since the source and destination overlap for copies. Note that the split cost is just part of the total insertion cost. Another part is the search cost for locating the right leaf node. The split cost is relatively independent of the depth of the tree since most of the splits happen on the leaves only. However, as the tree gets bigger, the search cost will increase in proportion to the depth of the tree. Although CSB^+-Trees have higher split cost than B^+-Trees, the total insertion cost will depend on the size of the tree.

Table 3 lists the space requirements of the various

Method	Internal Node Space	Typical Value	Leaf Node Space	Typical Value
B^+-Trees	$\frac{4nc}{0.7(m-2)(0.7m-2)}$	28.4 MB	$\frac{2nc}{0.7(m-2)}$	130.6 MB
CSB^+-Trees	$\frac{2nc}{0.7(m-2)(0.7m-1)}$	12.8 MB	$\frac{2nc}{0.7(m-2)}$	130.6 MB
CSB^+-Trees (t segments)	$\frac{2nc}{0.7(m-2)(0.7(m-2t)-0.3)}$	16.1 MB	$\frac{2nc}{0.7(m-2)}$	130.6 MB
Full CSB^+-Trees	$\frac{2nc}{(0.7)^2(m-2)(0.7m-1)}$	18.3 MB	$\frac{2nc}{(0.7)^2(m-2)}$	186.6 MB

Table 3: Space Analysis

algorithms, assuming all nodes are 70% full [Yao78]. We measure the amount of space taken by internal nodes and leaf nodes separately. We assume that each leaf node includes 2 sibling pointers. The internal space is calculated by multiplying $\frac{1}{q-1}$ (where q is the branching factor) by the leaf space. We do not include CSS-Trees in this comparison because CSS-Trees can never be "partially" full.

5 Experimental Results

We perform an experimental comparison of the algorithms on two modern platforms. The time we measured is the wall-clock time. We summarize our experiments in this section.

Experimental Setup. We ran our experiments on an Ultra Sparc II machine (296MHz, 1GB RAM) and a Pentium II (333MHz, 128M RAM) personal computer.[3] The Ultra machine has a <16k, 32B, 1> (<cache size, cache line size, associativity>) on-chip cache and a <1M, 64B, 1> secondary level cache. The PC has a <16k, 32B, 4> on-chip cache and a <512k, 32B, 4> secondary level cache. Both machines are running Solaris 2.6. We implemented all the methods including CSS-Trees, B^+-Trees, CSB^+-Trees, segmented CSB^+-Trees, and Full CSB^+-Trees in `C`. B^+-Trees, CSB^+-Trees, segmented CSB^+-Trees and Full CSB^+-Trees all support bulkload, search, insertion and deletion. We implemented "lazy" deletion since it's more practically used. CSS-Trees support only bulkload and search.

We choose keys to be 4-byte integers. For longer data types, we can put all distinct values in an order-preserving *domain* [Eng98] and use domain IDs as the keys. We also assume a TID and a pointer each takes four bytes. All keys are chosen randomly within the range from 1 to 10 million. The keys for various operations are generated randomly in advance to prevent the key generating time from affecting our measurements. We repeated each test three times and report the minimal time. When there are duplicates, the leftmost match is returned as the search result.

The Ultra Sparc processors provide two counters for event measurement [Inc99]. We used *perfmon*, a tool provided by Michigan State University [Enb99], to collect certain event counts and then calculate the number of secondary level cache misses.

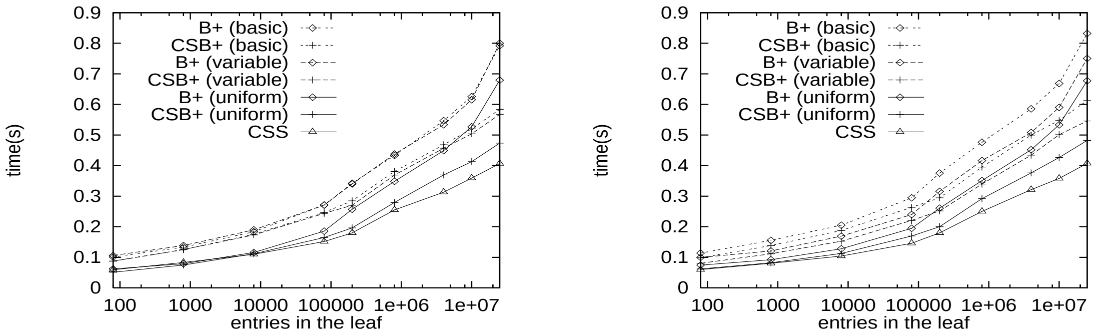
Implementation Details. As shown in [RR99], choosing the cache line size to be the node size is close to optimal for B^+-Trees. Thus we choose Ultra Sparc's cache line size to be the node size for all the searching methods. CSS-Trees have 16 keys per node. For B^+-Trees, each internal node consists of 7 keys, 8 child pointers, and the number of keys used. Each internal node for CSB^+-Trees consists of 14 keys, a first child pointer, and the number of keys used. Full CSB^+-Trees have the same node structure as CSB^+-Trees. We implemented segmented CSB^+-Trees with 2 segments and 3 segments. The 2-segment one has 13 keys and 2 child pointers per internal node whereas the 3-segment one has 12 keys and 3 child pointers per internal node (we use 1 byte to represent the size of each segment). For a 64-byte node size, it doesn't make sense to have more than 3 segments per node.

A leaf node in a B^+-Tree consists of 6 <key, TID> pairs, a forward and a backward pointer, and the number of entries being used. For CSB^+-Trees, all the nodes in a node group are stored contiguously. So, we don't really need to store sibling pointers for all the middle nodes in a group. For the first node and last node in a group, we need to store a forward pointer and a backward pointer respectively. As a result, we can squeeze 7 <key, TID> pairs in a leaf node. This optimization improves CSB^+-Tree's insertion performance by 10% and also reduces the amount of space needed.

For each method, we have three versions of the implementation corresponding to the basic, uniform and variable approaches described in Section 3.1.2. We use `#ifdef` in the code for the different code fragments among the versions. As a result, a lot of the code can be shared in the implementation.

During the bulkload, the higher level internal

[3] We omit the results for Pentium PC since they are similar to that for Ultra Sparc.



(a) Sun's `CC` (b) GNU's `gcc`

Figure 6: 200K Searches after Bulkload

nodes have to be filled with the largest keys in each subtree. We make this process more efficient by propagating the largest value in each subtree all the way up using the unused slots in each node. When building the higher level nodes, the last node could have only one child left from the lower level. In this case, we have no keys to put in the higher node. We address this problem by borrowing a key (and also the corresponding child) from the left sibling of the higher node. We implemented search and deletion iteratively to avoid unnecessary function calls. Insertion is still implemented recursively because of the difficulty of handling splits.

We implemented a simplified memory manager. Space is allocated from a large memory pool. Deallocated space is linked to a free pool. The space from the free pool could be used for other purposes although we didn't make use of it. We also didn't try to coalesce the free space since we expect this is done only occasionally and the cost is amortized.

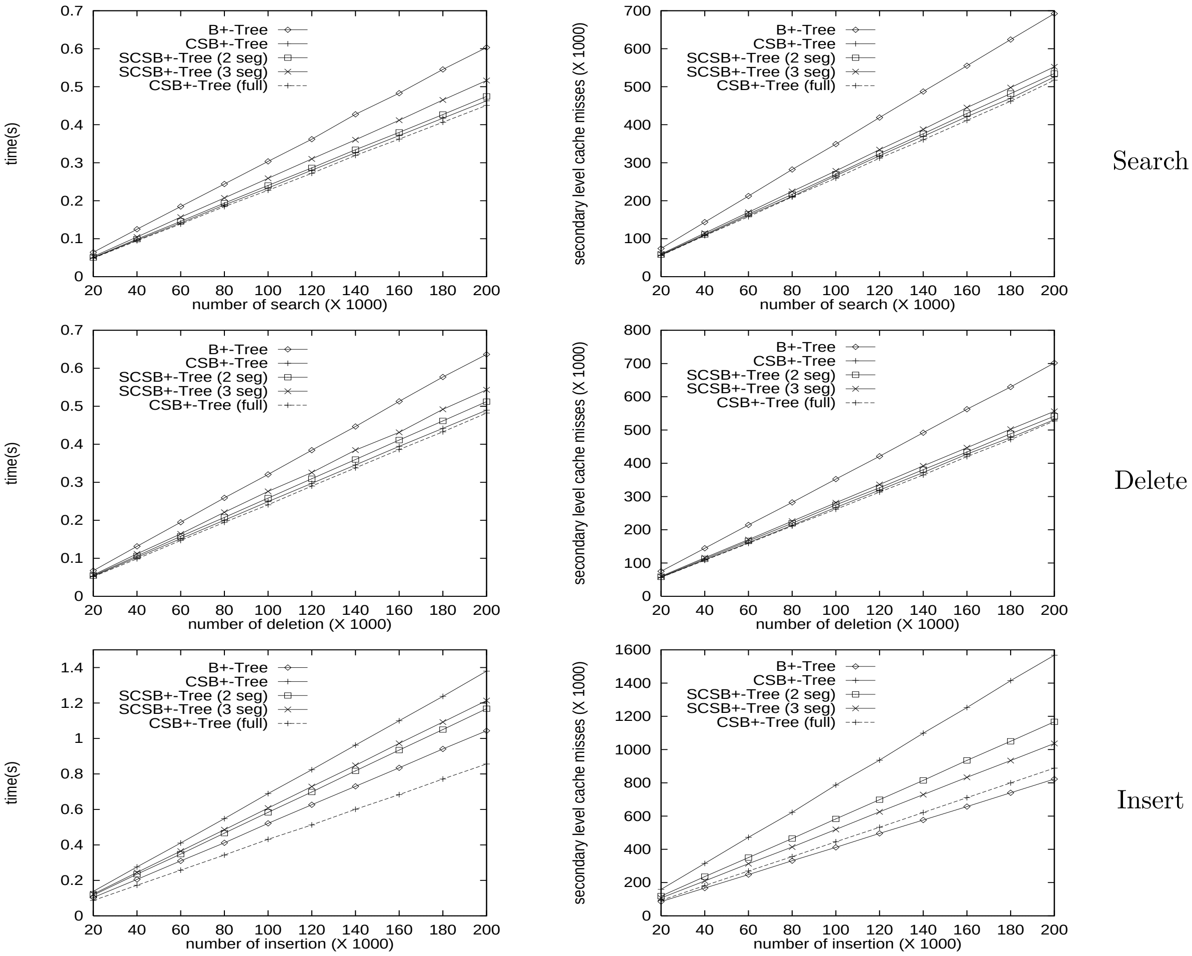
Since cache optimization can be sensitive to compilers [SKN94], we chose two different compilers: one is Sun's native compiler `CC` and the other is GNU's `gcc`. We used the highest optimization level of both compilers. Since we can't get the address of a label in Sun's `CC`, we use function arrays in the variant version for Sun's compiler. For `gcc`, we use its C extension of "Label as Values" [Pro99] and thus can eliminate the function calls.

Our implementations are specialized for a node size of 64 bytes. We use logical shifts in place of multiplication and division whenever possible. All the nodes are aligned properly according to the cache line size. Again, this is done on all the methods we are testing.

Results. In the first experiment, we want to compare the "pure" searching performance of various methods. We vary the number of keys in the leaf nodes during bulkloading. We measure the time taken by 200,000 searches. For B^+-Trees and CSB^+-Trees, we use all the slots in the leaf nodes and all the slots except one in the internal nodes. We tested all three versions of B^+-Trees and CSB^+-Trees. Figure 6(a) and 6(b) show the result using Sun's `CC` and `gcc` respectively. CSS-Trees are the fastest. Besides having a larger branching factor, CSS-Trees can put 8 <key, TID> pairs in the leaf nodes since it assumes the leaves are kept in a sorted array. CSB^+-Trees perform slightly worse than CSS-Trees. B^+-Trees are more than 25% slower than CSB^+-Trees. Among the three versions we tested, the uniform approach performs the best for both compilers. The variable approach using Sun's `CC` is actually a little bit worse than the basic one. This is because of the overhead introduced by functions calls. When function call overhead is removed, as shown in Figure 6(b), the variable version performs better than the basic one. However, the variable version is still worse than the uniform version. There are two reasons. First, there is an extra jump instruction in the variable version. Second, when the nodes are almost full, the variable version uses almost the same number of comparisons as the uniform version. Since the pattern among the versions is the same across all tests, we only present the result of the uniform version (using Sun's CC) in the remaining sections.

In our next experiment, we test the individual performance of search, insertion and deletion when the index structure stabilizes. To simulate that, we first bulkload 0.4 million entries followed by inserting 3.6 million new entries. We then perform up to 200,000 operations of each kind and measure their time. Figure 7 shows the elapsed time and the number of secondary level of cache misses.

For searching, CSB^+-Trees perform better than B^+-Trees as expected. CSB^+-Trees better utilize each cache line and thus have fewer cache misses than B^+-Trees as verified by our cache measurement. The larger the number of searches, the wider



(a) Time (b) Secondary Level Cache Misses

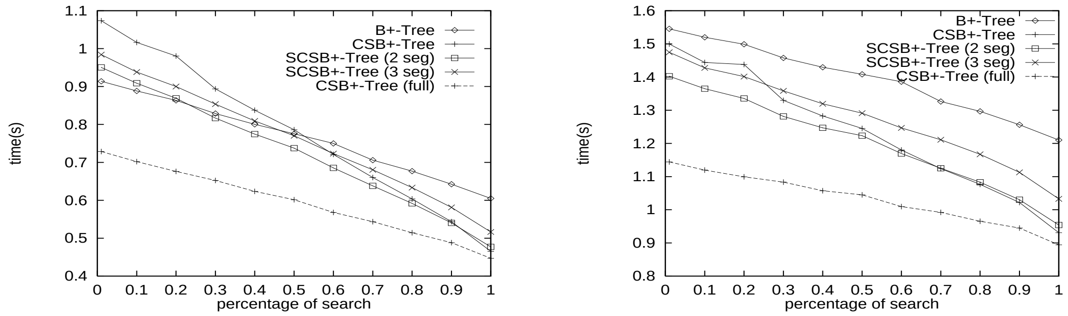
Figure 7: 200K Operations on a Stabilized Index Structure

the gap between the two. Segmented CSB$^+$-Trees fall between CSB$^+$-Trees and B$^+$-Trees. There are two reasons why segmented CSB$^+$-Trees search slower than CSB$^+$-Trees. First, the branching factor for segmented CSB$^+$-Trees is less since we have to record additional child pointers. This causes segmented CSB$^+$-Trees to have slightly more cache misses than CSB$^+$-Trees. Second, extra comparisons are needed to choose the right segment during tree traversal. Nevertheless, 2-segment CSB$^+$-Trees perform almost as well as CSB$^+$-Trees. Full CSB$^+$-Trees perform a little bit better than CSB$^+$-Trees and have fewer cache misses. We suspect this is because the nodes in full CSB$^+$-Trees are aligned in a way that reduces the number of conflict cache misses. Unfortunately, we can't distinguish between a conflict miss and a capacity miss using the current counter events.

The delete graph is very similar to that of search. This is because in "lazy" deletion, most of the time is spent on locating the correct entry in the leaf. Delete takes a little bit more time than search since we may have to walk through several leaf nodes to find the entry to be deleted.

CSB$^+$-Trees are worse than B$^+$-Trees for insertion. The insertion cost has two parts, one is the search cost and the other is the split cost. The split cost of CSB$^+$-Trees includes copying a complete node group, whereas that of B$^+$-Trees is creating a single new node. In our test, we observe there are about 50,000 splits (one every four inserts).[4] As a result, CSB$^+$-Trees take more time to insert than B$^+$-Trees. Segmented CSB$^+$-Trees reduce the split cost. Now the copying unit is a segment. When nodes are relatively evenly distributed across segments, the copying cost is reduced. That's why we see 2-segment CSB$^+$-Tree performing better than CSB$^+$-Trees. The 3-segment CSB$^+$-Tree is no better than the 2-segment one. The reason is that it's hard to distribute fewer than 12 keys

[4] This is consistent with the estimate of the average number of splits per insertion ($\frac{1}{1.386d}$) in [Wri85].



(a) 4 million data (b) 16 trillion data (estimate)

Figure 8: Varying Workload on a Stabilized Index Structure

evenly among 3 segments. Large segments take more time to copy and are more likely to be selected for insertion. Additionally, more segments means extra comparisons during the search. An important issue is that while the split cost is relatively fixed (since most of the splits are on the leaves), the search cost depends on the size of the tree. The larger the data set, the higher the search cost. So the insertion cost will be different (favoring CSB$^+$-Trees) when the indexed data is much larger.

Full CSB$^+$-Trees perform insertion much faster than CSB$^+$-Trees. This observation was predicted in Section 3.3. What's even more interesting is that full CSS-Trees are even better than B$^+$-Trees on insert. The number of cache misses doesn't explain the difference since full CSB$^+$-Trees have more cache misses. It's likely that the explanation is that the allocation overhead for full CSB$^+$-Trees is lower. B$^+$-Trees have to allocate a new node on every split while full CSB$^+$-Trees make an allocation only when a node group is full.

Our last experiment tests the overall performance of all the methods. We first build the same stabilized tree as in the previous experiment and then perform 200,000 operations on it. We vary the percentage of searches and fix the ratio between inserts and deletes to be 2:1. The result is shown in Figure 8(a). Full CSB$^+$-Trees perform the best across the board. However, it uses somewhat more space than other methods. At the left end, B$^+$-Trees perform better than all but Full CSB$^+$-Trees. As more and more searches are performed, the cost of all the CSB$^+$-Trees decreases much faster than B$^+$-Trees. CSB$^+$-Tree starts to perform better than B$^+$-Tree when more than 45% of the operations are searches. 2-segment CSB$^+$-Tree is better than both CSB$^+$-Tree and B$^+$-Tree when the percentage of searches is between 25% and 90%.

To see how the cost of the methods scales with data size, we estimate the cost of all the methods under a much larger data set. The search cost increases in proportion to the data size, while the split cost remains roughly the same. We separate the time in Figure 8(a) into two parts: search and split. We then scale the search time proportionally and combine it with the unchanged split time. Figure 8(b) shows the result when the search cost is doubled (corresponding to 16 Trillion of leaf entries). As we can see, all variants of CSB$^+$-Trees win across the board. Note that we're not claiming that trillions of data items is realistic for main memory in the near future. The point of Figure 8(b) is to show the limiting behavior, and to illustrate that as the data gets bigger, the performance of the various CSB$^+$-Trees improves relative to B$^+$-Trees due to the increased dependence of overall performance on search time.

5.1 Summary

Full CSB$^+$-Trees are better than B$^+$-Tree in all aspects except for space. When space overhead is not a big concern, Full CSB$^+$-Tree is the best choice. When space is limited, CSB$^+$-Trees and segmented CSB$^+$-Trees provide faster searches while still able to support incremental updates. Many applications, such as online shopping and digital libraries that we described in Section 1, have many more searches than updates (inserts, to be more accurate). For those applications, CSB$^+$-Trees and segmented CSB$^+$-Trees are much better than B$^+$-Trees. Depending on the workload, either of the CSB$^+$-Tree variants could be the best. We summarize the results in Table 4. Note that the ratings in the table are qualitative relative judgments. The precise numerical values for relative performance can be found in the previous section.

Our experiments are performed for 4-byte keys and 4-byte child pointers. Theoretically, B$^+$-Trees will have 30% more cache misses than CSB$^+$-Trees. As we have seen, our implementation of CSB$^+$-Trees has achieved most of the benefit. In the

	B^+	CSB^+	$SCSB^+$	Full CSB^+
Search	slower	faster	medium	faster
Update	faster	slower	medium	faster
Space	medium	lower	lower	higher
Memory Management Overhead	medium	higher	higher	lower

Table 4: Feature Comparison

next generation operating systems, if both the key size and the pointer size double (assuming the same cache line size), B^+-Trees will have 50% more cache misses than CSB^+-Trees and we'd expect more significant improvement by using CSB^+-Trees.

We close the presentation of the experiments by noting that many of the performance graphs are architecture dependent. Changes in compiler optimization methods or in architectural parameters may affect the relative performance of the algorithms. Nevertheless, the fundamental reason why the various CSB^+-Trees win is that they are *cache sensitive*, getting better utilization of each cache line. We expect cache sensitivity to be even more critical as CPU speeds continue to accelerate much faster than RAM speeds.

6 Conclusion

In this paper, we proposed a new index structure called a CSB^+-Tree. CSB^+-Trees are obtained by applying partial pointer elimination to B^+-Trees. CSB^+-Trees utilize more keys per cache line, and are thus more cache conscious than B^+-Trees. Unlike a CSS-Tree, which requires batch updates, a CSB^+-Tree is a general index structure that supports efficient incremental updates. Our analytical and experimental results show that CSB^+-Trees provide much better performance than B^+-Trees in main memory because of the better cache behavior. As the gap between CPU and memory speed is widening, CSB^+-Trees should be considered as a replacement for B^+-Trees in main memory databases. Last but not least, partial pointer elimination is a general technique and can be applied to other in-memory structures to improve their cache behavior.

References

[ADW99] Anastassia Ailamaki, et al. DBMSs on a modern processor: Where does time go. In *Proceedings of the 25th VLDB Conference*, 1999.

[BBC+98] Phil Bernstein, et al. The Asilomar report on database research. *Sigmod Record*, 27(4), 1998.

[BMK99] Peter A. Boncz, et al. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th VLDB Conference*, 1999.

[CLH98] Trishul M. Chilimbi, et al. Improving pointer-based codes through cache-conscious data placement. Technical report 98, University of Wisconsin-Madison, Computer Science Department.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surverys*, 11(2), 1979.

[Enb99] Richard Enbody. Permon performance monitoring tool (available from http://www.cps .msu.edu/~enbody/perfmon.html). 1999.

[Eng98] InfoCharger Engine. Optimization for decision support solutions (available from http:// www.tandem.com/prod_des/ifchegpd/ifchegpd.htm). 1998.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a quantitative approach*. Morgan Kaufman, San Francisco, CA, 2 edition, 1996.

[Inc99] Sun Microsystems Inc. Ultra sparc user's manual (available from http://www.sun.com/ microelectronics/manuals/ultrasparc/802-7220-02.pdf as of oct. 16, 1999). 1999.

[Ker89] Martin L. Kersten. Using logarithmic code-expansion to speedup index access and maintenance. In *Proceedings of 3rd FODO Conference*, pages 228–232, 1989.

[LC86] Tobin J. Lehman, et al. A study of index structures for main memory database management systems. In *Proceedings of the 12th VLDB Conference*, 1986.

[O'N92] Patrick E. O'Neil. The SB-tree: An index-sequential structure for high-performance sequential access. *Acta Informatica*, 29(3):241–265, 1992.

[Pro99] GNU Project. Gun c compiler manual (available from http://www.gnu.org/software/gcc/ onlinedocs/gcc_toc.html as of oct. 16, 1999). 1999.

[Ram97] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1997.

[RR99] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th VLDB Conference*, 1999.

[SKN94] Ambuj Shatdal, et al. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th VLDB Conference*, 1994.

[Smi82] Alan J. Smith. Cache memories. *ACM Computing Surverys*, 14(3):473–530, 1982.

[TMJ98] Kristian Torp, et al. Efficient differential timeslice computation. *IEEE Transactions on knowledge and data engineering*, 10(4), 1998.

[Wri85] William Wright. Some average performance measures for the B-tree. *Acta Informatica*, 21:541–557, 1985.

[Yao78] Andrew Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978.