Cache Conscious Indexing for Decision-Support in Main Memory

Jun Rao
Columbia University
junr@cs.columbia.edu

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

Abstract

We study indexing techniques for main memory, including hash indexes, binary search trees, T-trees, B+-trees, interpolation search, and binary search on arrays. In a decision-support context, our primary concerns are the lookup time, and the space occupied by the index structure.

Our goal is to provide faster lookup times than binary search by paying attention to reference locality and cache behavior, without using substantial extra space. We propose a new indexing technique called "Cache-Sensitive Search Trees" (CSS-trees). Our technique stores a directory structure on top of a sorted array. Nodes in this directory have size matching the cache-line size of the machine. We store the directory in an array and do not store internal-node pointers; child nodes can be found by performing arithmetic on array offsets.

We compare the algorithms based on their time and space requirements. We have implemented all of the techniques, and present a performance study on two popular modern machines. We demonstrate that with a small space overhead, we can reduce the cost of binary search on the array by more than a factor of two. We also show that our technique dominates B+-trees, T-trees, and binary search trees in terms of both space and time. A cache simulation verifies that the gap is due largely to cache misses.

*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.

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

1 Introduction

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 The traditional assumption that memory references have uniform cost is no longer valid given the current speed gap between cache access and main memory access. So, improving cache behavior is going to be an imperative task in main memory data processing. In this paper, we focus on how to make indexes cache conscious.

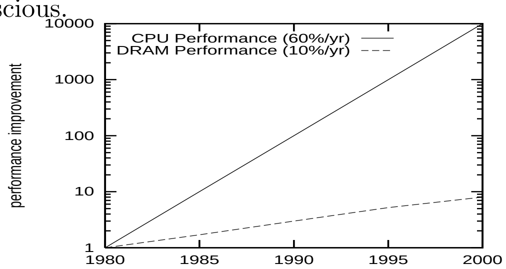
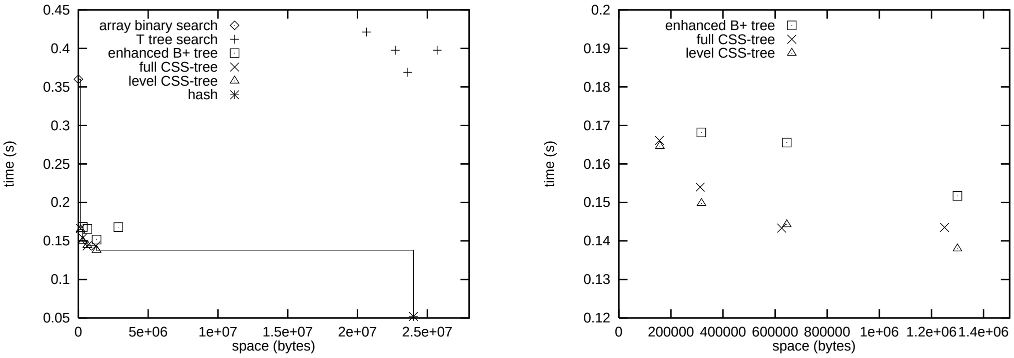


Figure 1: Processor-memory performance imbalance

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. With a large amount of RAM, most of the indexes can be memory resident. Past work



(a) Whole Picture (b) Close-up

Figure 2: Space/time Tradeoffs

on measuring the performance of indexing in main-memory databases includes [LC86b, WK90], with [LC86b] being the most comprehensive on the specific issue of indexing. In the thirteen years since [LC86b] was published, there have been substantial changes in the architecture of computer chips. The most relevant change is that CPU speeds have been increasing at a much faster rate (60% per year) than memory speeds (10% per year) as shown in Figure 1 (borrowed from [CLH98]). Thus, the *relative* cost of a cache miss has increased by two orders of magnitude since 1986. As a result, we cannot assume that the ranking of indexing algorithms given in [LC86b] would be valid on today's architectures. In fact, our experimental results indicate very different relative outcomes from [LC86b] for lookup speed.

Another recent development has been the explosion of interest in On-Line Analytical Processing (OLAP). [Fre95, Fre97] contrasts the requirements of OLAP with OLTP systems and contends that the real performance gains can be obtained by separating the two systems. A dedicated OLAP system can have a much better query performance if we are willing to sacrifice update performance. Commercial systems such as Sybase IQ [Syb97] were designed for such purposes. Thus, typical OLAP workloads are query-intensive, and have infrequent batch updates. For example, a data warehouse for a university, containing all the student records, is probably updated once per day. Census data sets are collected periodically and will remain static for a relatively long period of time. These systems are in the scale of several gigabytes and can already fit in RAM today. Given the current trend, we expect more and more disk-based applications will be moved to main memory in the future. Since updates are batched in those systems, incremental updates of indexes may not be very important. In fact, we may want to rebuild indexes from scratch after a batch of updates, if that leads to faster index searches. In this paper, we focus on such an OLAP environment.

Two important criteria for the selection of index structures are *space* and *time*. Space is critical in a main memory database; we may have a limited amount of space available for precomputed structures such as indexes. Given space constraints, we try to optimize the time taken by index lookups. In a main-memory database there are several factors influencing the speed of database operations. An important factor is the degree of locality in data references for a given algorithm. Good data locality leads to fewer (expensive) cache misses, and better performance.

We study a variety of existing techniques, including hash indexes, binary search on a sorted list of record identifiers, binary trees, B+-trees [Com79], T-trees [LC86a], and interpolation search. We also introduce a new technique called "Cache-Sensitive Search Trees" (CSS-trees). CSS-trees augment binary search by storing a directory structure on top of the sorted list of elements. CSS-trees differ from B+-trees by avoiding storing the child pointers in each node. The CSS-tree is organized in such a way that traversing the tree yields good data reference locality (unlike binary search), and hence relatively few cache misses. CSS-trees also take advantage of the OLAP context to optimize search performance, at the expense of update performance.

We summarize the space/time tradeoffs of various methods in Figure 2. Each point for T-trees, enhanced B+-trees and CSS-trees corresponds to a specific node size (multiple of cache line size). Normally a larger node size means less space but more search time. The stepped line basically tells us what's the optimal searching time for a given amount of space. Our conclusion is that CSS-trees dominate T-trees and enhanced B+-trees in both space and time. There are tradeoffs between space and time for binary search, CSS-trees and hash indices. CSS-trees reasonably balance space and time. We discuss this issue in more detail in Section 7.

2 Main Memory Databases

Main Memory Database Systems: Some past work on main memory databases has addressed the problems of concurrency, transaction processing and logging [GMS86, LN88, JLRS94], and recovery [Hag86, LC87]. Systems with a significant query-processing component include OBE [WK90], MM-DBMS [LC87], and Starburst [LSC92]. More recently, the TimesTen corporation (formerly the Smallbase project at Hewlett-Packard) has developed a commercial main-memory system, with claims of tenfold speedups over disk-based systems [Sof97]. Most of the systems do not address the issue of cache reference locality.

Data Layout: [AHK85] and others describe the concept of a domain. When data is first loaded into main memory, distinct data values are stored in an external structure, the domain, and only pointers to domain values are stored in place in each column. This has the benefits of: (a) saving space in the presence of duplicates, (b) simplified handling of variable-length fields and (c) pointer comparisons can be used for equality tests. In the main memory database project at Columbia University, we focus on an OLAP main memory database system. We go further than [AHK85] by keeping the domain values in order and associate each value with a domain ID (represented by an integer). As a result, we can process both equality and inequality tests on domain IDs directly, rather than on the original values. Although keeping values in order has extra cost, we expect the data is updated infrequently. Independently, Tandem Inc.'s InfoCharger storage engine [Eng98] has also chosen to keep domain values in sorted order. The use of domains means that many indexes can be built with smaller keys.

Indexing in Main Memory Databases: Although sequential data access is much cheaper in main memory than in disk-based systems, indexing remains very important in main memory databases. First of all, searching an index is still useful for answering single value selection queries and range queries. Next, cheaper random access makes indexed nested loop joins more affordable in main memory databases. Indexed nested loop join is pipelinable, requiring minimal storage for intermediate results and is relatively easy to implement. As a matter of fact, indexed nested loop join is the only join method used in [WK90]. This approach requires a lot of searching through indexes on the inner relations. Last but not least, transforming domain values to domain IDs (as described in the previous section) requires searching on the domain.

A list of record identifiers sorted by some columns provides ordered access to the base relation. Ordered access is useful for range queries and for satisfying interesting orders [SAC+79]. A sorted array is an index structure itself since binary search can be used.

Assumptions: We assume an OLAP environment, so we don't care too much about updates. Our main concerns are the lookup time for an index, and the space required to store the index. There are many applications in this class, as described in Section 1.

3 Cache Optimization on Indexes

In this section, we first describe cache memories and the impact of cache optimization. We then give a survey of the related work. Finally, we analyze the cache behavior of various existing index structures for searching and point out their shortcomings.

Cache memories are small, fast static RAM memories that improve program performance by holding recently referenced data [Smi82]. 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.

A cache can be parameterized by capacity, block size and associativity, where capacity is the size of the cache, block size is the basic transferring unit between cache and main memory, and associativity determines how many slots in the cache are potential destinations for a given address reference.

Cache optimization in a main memory database system is similar to main memory optimization in a disk-based system. But the management of the cache is done by the hardware and the database system doesn't have direct control of which block to bring into a cache. This makes cache optimization more subtle.

Typical cache optimization techniques include clustering, compression and coloring [CLH98]. Clustering tries to pack, in a cache block, data structure elements that are likely to be accessed successively. Compression tries to remove irrelevant data and thus increases cache block utilization by being able to put more useful elements in a cache block. This includes key compression, structure encodings such as pointer elimination and fluff extraction. Caches have finite associativity, which means that only a limited number of concurrently accessed data elements can map to the same cache line without causing conflict. Coloring maps contemporaneously-accessed elements to non-conflicting regions of the cache.

Previous research has attacked the processor-memory gap using the above techniques. Wolf and Lam [WL91] exploited cache reference locality to improve the performance of matrix multiplication. LaMarca and Ladner [LL96, LL97] considered the effects of caches on sorting algorithms and improved performance by restructuring these algorithms to exploit caches. In addition, they constructed a cache-conscious heap structure that clustered and aligned heap elements to cache blocks. [CLH98] demonstrated that cache optimization techniques can improve the spatial and temporal locality of pointer-based data structures. They showed improvement on various benchmarks.

In [NBC$^+$94], Nyberg et al. have shown that for achieving high performance sorting, one should focus carefully on cache memory behavior.

Cache conscious algorithms have been considered in database systems also. In [SKN94], the authors suggested several ways to improve the cache reference locality of query processing operations such as joins and aggregations. They showed that the new algorithms can run 8% to 200% faster.

Although cache optimization has been considered on tree-based structures, nobody has looked at the influence of the cache on index structures used in database systems. Some other papers have considered the issue of compact representations of B-tree indexes [CS83, JTR87]. These papers appeared too early to consider cache issues since memory speed wasn't too slow compared with CPU speed. In the rest of this section, we study the cache behavior of various typical index structures used in main memory database systems.

Array Binary Search: The problem with binary search is that many accesses to elements of the sorted array result in a cache miss. We do not get misses for the first references because of temporal locality over many searches. We avoid misses for the last references, due to spatial locality, if many records from the array fit inside a single cache line. However, when the array is substantially bigger than the cache, many of the intervening accesses cause cache misses. In the worst case, the number of cache misses is of the order of the number of key comparisons.

T-Trees: T-Trees have been proposed as a better index structure in main memory database systems. A *T-Tree* [LC86a] is a balanced binary tree with many elements in a node. Elements in a node contain adjacent key values and are stored in order. Its aim is to balance the space overhead with searching time and cache behavior is not considered (thirteen years ago the gap between processor and main memory speeds was not that large). T-Trees put more keys in each node and give the impression of being cache conscious. But if we think carefully, we can observe that for most of the T-Tree nodes, only the two end keys are actually used for comparison (in the improved version [LC86b], only one key is used). This means that the utilization of each node is low. Since the number of key comparisons is still the same, T-Trees do not provide any better cache behavior than binary search.

Another problem with the T-Tree is that it has to store a record pointer for each key within a node. Since most of the time the record pointers won't be needed, essentially half of the space in each node is wasted. Potentially, one could put just RIDs in the T-tree with no keys, but then search is much slower due to indirection.

B+-trees: Although B+-trees were designed for disk-based database systems, they actually have a much better cache behavior than T-trees. In each internal node we store keys and child pointers, but the record pointers are stored on leaf nodes only. Multiple keys are used to search within a node. If we fit each node in a cache line, this means that a cache load can satisfy more than one comparison. So each cache line has a better utilization ratio.

Enhanced B+-trees: In an OLAP environment, we can use all the slots in a B+-tree node (similar to compact B-Trees [CS83] and the ISAM method used in the IBM OS/360 operating system [GR93]) and rebuild the tree when batch updates arrive. We can design the node size to be exactly the same as a cache line and align each node. We can also hard-code the node search since we know all the slots are used. But B+-trees enhanced in these ways still need to store child pointers within each node (even when compact). So for any given node size, only half of the space can be used to store keys.

In the rest of this paper we shall consider only "enhanced B+-trees" as described above; however, we may sometimes use "B+-trees" for brevity.

Hash: Hash indices can also benefit from cache optimization. The most common hashing method is chained bucket hashing [Knu73]. In [GBC98], the authors use the cache line size as the bucket size and squeeze in as many $< key, RID >$ pairs as possible. This can reduce the number of cache misses when scanning through the buckets. Hash indices are fast for searching only if the length of each bucket chain is small. This requires a fairly large directory size and thus a fairly large amount of space. Skewed data can seriously affect the performance of hash indices unless we have a relatively sophisticated hash function, which will increase the computation time. Hash indices do not preserve order. To provide ordered access, an ordered list has to be kept in addition to the hash indices.

4 Cache Sensitive Search Trees

In this section, we present our cache conscious searching methods called CSS-trees. Section 4.1

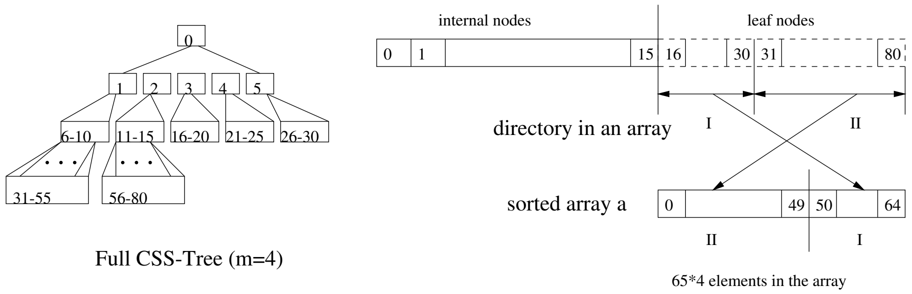


Figure 3: Layout of a full CSS-tree

introduces the concept of "full" CSS-trees. We talk about "level" CSS-trees in Section 4.2.

Suppose that we have a sorted array `a[1..n]` of `n` elements. The array `a` could contain the record-identifiers of records in some database table in the order of some attribute k. `a` could alternatively contain column-k keys from the records in the table, with a companion array holding the corresponding record-identifiers, using some extra space to avoid an indirect reference during the search. `a` could alternatively contain records of a table or domain values.

Binary search of `a` has a serious cache usage problem as described in Section 3. A second problem with binary search is that it requires a calculation to be performed $\log_2 n$ times to find the next element to search. Even if this calculation uses a shift rather than a division by two [WK90], the calculation represents a significant portion of the execution time needed. Nevertheless, binary search has the benefit that no additional space beyond `a` is needed to perform a search. Our goal is to improve upon the search time of binary search without using a significant amount of additional space.

4.1 Full CSS-Trees

We create a search tree with nodes containing exactly m keys. (We'll see how to choose m later.) If the depth of the tree is d, then the tree is a complete $(m+1)$-ary search tree up to depth $d-1$, and at depth d the leaves are filled from left to right. An example tree is shown for $m = 4$ in the left diagram of Figure 3 (the numbers in the boxes are node numbers and each node has four keys). The nodes of a CSS-tree can be stored in an array as shown on the right in Figure 3. Note that we do not need to store explicit pointers to child nodes; the children of a node can be computed from offsets in the CSS-tree array. (The exact formulas are given below.)

Our basic idea is to store a CSS-tree as a directory structure on top of the array `a`. There are two reasons why we expect a traversal of such a tree to be faster than binary search. First, if we choose m such that a node fits in a cache line, then all local searching within a node happens with at most one cache miss. As a result we get at most $\log_{m+1} n$ cache misses for a lookup, unlike binary search which gets up to $\log_2 n$ cache misses. (Even if a node occupies two cache lines, half the time only one cache miss will be generated, while half the time there will be two cache misses.) Second, we can hard-code the traversal within a node, so that calculations needed to find the next node happen $\log_{m+1} n$ times rather than $\log_2 n$ times. (Note that the total number of comparisons is the same.)

Suppose that we number the nodes and keys starting at 0. If we have an internal node numbered b, then it is not difficult to show that the children of that node are numbered from $b(m+1)+1$ to $b(m+1)+(m+1)$. Within the directory, we have m keys per node, so key number i in the directory array maps to node number $\lfloor \frac{i}{m} \rfloor$. As a result, one can find the offset of the first key of the child nodes within the directory array as $(\lfloor \frac{i}{m} \rfloor * (m+1)+1) * m$ through $(\lfloor \frac{i}{m} \rfloor * (m+1)+m+1) * m$.

A subtle point in the structure of a CSS-tree is that we store the leaf nodes in a contiguous array in key order. This conflicts with the natural order of the CSS-tree that stores the nodes from left to right, level by level. The CSS-tree order would split the array, putting the right half of the array (which appears at a higher level than the left half of the array) ahead of the left half of the array. In Figure 3, the natural tree order is to store nodes 16-30 before nodes 31-80. However, when the leaves are stored in a sorted array, nodes 31-80 come before nodes 16-30. Since maintaining a contiguous array in key order is desirable for other purposes, and since the array is given to us without assumptions that it can be restructured, we leave the array in key order. To get to the correct leaves when searching the CSS-tree, we modify the natural search algorithm.

When performing a search, we move from parent to child by recalculating the offset within the directory structure as above. We mark the end of the directory structure (in Figure 3, that's the last key in node 15), and terminate this portion of the search when the computed offset exceeds the endpoint. If the leaves were stored in the natural CSS-tree order, we'd use this offset to look up the directory directly.

However, since the two parts of the leaf nodes are stored in reverse order in a separate array, we need to process this offset further.

The two parts of the leaf nodes are mapped into the sorted array as shown in Figure 3. We use y to denote the boundary of the two parts of the leaf nodes, which is the offset of the first key at the deeper leaf level in the directory array. (In Figure 3 that's the first key in node 31.) Given an offset x of a key in a leaf node, we compare it with y to determine which part of the sorted array to look into. If $x > y$, we find the element at position $x - y$ from the *start* of the sorted array a. Otherwise, we find the element at position $y - x$ from the *end* of a. For example, in Figure 3, the first key in leaf node 30 can be found at the first key in node 64 in the sorted array.

Note that our techniques apply to sorted arrays having elements of size different from the size of a key. Offsets into the leaf array are independent of the record size within the array; the compiler will generate the appropriate byte offsets.

The following lemma tells us how to calculate the node number of the last internal node, and the node number y. Key offsets can be obtained by multiplying these numbers by m.

Lemma 4.1: Let $n = N * m$ be the number of elements in the sorted array a (N is the number of leaf nodes). The total number of internal nodes in a full CSS-tree is $\frac{(m+1)^k-1}{m} - \lfloor\frac{(m+1)^k-N}{m}\rfloor$. The first leaf node in the bottom level is number $\frac{(m+1)^k-1}{m}$. In both formulas, $k = \lceil\log_{m+1}(N)\rceil$.

Notice that CSS-trees are stored in a way similar to heaps. This is possible because of the way we "virtually" split the sorted array. B+-trees can't use the same technique since they require all the leaves to be on the same level.

Building a Full CSS-Tree: To build a full CSS-tree from a sorted array, we first split the sorted array *logically* into two parts (based on Lemma 4.1) and establish the mapping between the leaf nodes and elements in the sorted array. We then start with the last internal node. For each entry in the node, we fill it with the value of the largest key in its immediate left subtree. Finding the largest key in a subtree can be done by following the link in the rightmost branch until we reach the leaf nodes.

Some internal nodes, namely ancestors of the last leaf node at the deepest level, may not have a full complement of keys. In our algorithm, we simply fill in those dangling keys with the last element in the first half of array a. So there may be duplicate keys in some internal nodes. In our searching algorithm, we tune the searching within each node in such a way that the leftmost key will always be found in case of duplicates. So we will never reach the leaf nodes in the deepest level with an index out of the range of the first half of the sorted array.

Although it's difficult to incrementally update a full CSS-tree, it's relatively inexpensive to build such a tree from scratch. Our experiments show that to build a full CSS-tree from a sorted array of twenty-five million integer keys takes less than one second on a modern machine. Therefore, when batch updates arrive, we can afford to rebuild the CSS-tree.

Searching a Full CSS-Tree: Once a full CSS-tree is built, we can search for a key. We start from the root node. Every time we reach an internal node, we do a binary search to find out which branch to go to. We repeat until we reach a leaf node. Finally, we map the leaf node into the sorted array and binary search the node.

All the searches within a node consist of hard-coded *if-else* statements. When doing binary search in the internal nodes, we keep checking whether the keys in the left part are greater than or equal to the searching key. We stop when we find the first slot that has a value smaller than the searching key and then follow the branch on the right. (If such a slot can't be found, we follow the leftmost branch.) In this way, if there are duplicates in a node, we are guaranteed to find the leftmost key among all the duplicates. When there are duplicate keys being indexed, we can return the leftmost match in a fashion similar to B+-trees.

4.2 Level CSS-Trees

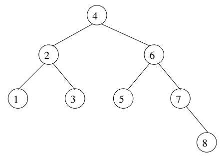


Figure 4: Node with 8 keys

A full CSS-tree with m entries per node will have exactly m keys, i.e., all the entries are fully used. Figure 4 shows the binary search tree of a node with $m = 2^3$ entries. Out of the nine possible branches, seven of them need three comparisons and two of them need four. But if we waste one entry and just put seven keys per node, we will have a full binary search tree and all the branches need three comparisons. This may give us some benefit. So for $m = 2^t$, we define a tree that only uses $m-1$ entries per node and has a branching factor of m a *level CSS-tree*. A level CSS-tree will be deeper than the corresponding full CSS-tree since now the branching factor is m instead of $m+1$. However, we have fewer comparisons per node. If N is the number of nodes that the elements in the sorted array can form, a

Method	branching factor	# of levels (l)	comparisons per internal node (nComp)	comparisons per leaf node (A_{child})
Binary search	2	$\log_2 n$	1	1
T-trees	2	$\log_2 \frac{n}{m} - 1$	1	$\log_2 m$
enhanced B+-trees	$\frac{m}{2}$	$\log_{\frac{m}{2}} \frac{n}{m}$	$\log_2 m - 1$	$\log_2 m$
Full CSS-trees	$m+1$	$\log_{m+1} \frac{n}{m}$	$(1 + \frac{2}{m+1}) \log_2 m$	$\log_2 m$
Level CSS-trees	m	$\log_m \frac{n}{m}$	$\log_2 m$	$\log_2 m$

Method	Total comparisons	Moving across Level Level	Cache Misses $\frac{mK}{c} <= 1$	Cache Misses $\frac{mK}{c} > 1$
Binary search	$\log_2 n$	$\log_2 n * A_b$	$\log_2 n$	$\log_2 n$
T-trees	$\log_2 n$	$\log_2 n * D$	$\log_2 n$	$\log_2 n$
enhanced B+-trees	$\log_2 n$	$\log_{\frac{m}{2}} \frac{n}{m} * D$	$\frac{\log_2 n}{\log_2 m-1}$	$\log_{\frac{m}{2}} n((\log_2 \frac{mK}{c}) + \frac{c}{mK})$
Full CSS-trees	$\frac{m+3}{m+1} \log_{m+1} m \log_2 n$	$\log_{m+1} \frac{n}{m} * A_{fcss}$	$\frac{\log_2 n}{\log_2 (m+1)}$	$\log_{m+1} n((\log_2 \frac{mK}{c}) + \frac{c}{mK})$
Level CSS-trees	$\log_2 n$	$\log_m \frac{n}{m} * A_{lcss}$	$\frac{\log_2 n}{\log_2 m}$	$\log_m n((\log_2 \frac{mK}{c}) + \frac{c}{mK})$

Table 1: Time analysis

level CSS-tree has $\log_m N$ levels while a full CSS-tree has $\log_{m+1} N$ levels. The number of comparisons per node is t for a level CSS-tree and $t * (1 + \frac{2}{m+1})$ for a full CSS-tree. So the total number of comparisons for a level CSS-tree is $\log_m N * t = \log_2 N$ and that for a full CSS-tree is $\log_{m+1} N * t * (1 + \frac{2}{m+1}) = \log_2 N * \log_{m+1} m * (1 + \frac{2}{m+1})$. The ratio of the former to the latter is $\frac{(m+1)\log_m (m+1)}{m+3}$. Thus a level CSS-tree always uses fewer comparisons than a full CSS-tree for searching. On the other hand, level CSS-trees may require $\log_m N$ cache accesses and $\log_m N$ node traversals, compared with $\log_{m+1} N$ for full CSS-trees. Whether we obtain a net gain in speed depends upon how time-consuming a comparison operation is compared with node traversals and cache accesses. A level CSS-tree still utilizes most of the data in each cache line. It uses a little more space than a full CSS-tree, but this may be desirable for users who want to trade space for time.

The building of level CSS-trees is similar to that of full CSS-trees. We can also make good use of the empty slot in each node. During the population, we can use that slot to store the largest value in the last branch of each node. We can thus avoid traversing the whole subtree to find the largest element. The searching algorithm of a level CSS-tree is similar to that of a full CSS-tree. The only difference is the calculation of the offset of a child node.

5 Time and Space Analysis

In this section, we analytically compare the time performance and the space requirement for different methods. We let R denote the space taken by a record-identifier, K denote the space taken by a key, P denote the space taken by a child pointer and n denote the number of records being indexed. h denotes a hashing fudge factor (typically about 1.2, meaning that the hash table is 20% bigger than the raw data in the hash table). c denotes the size of a cache line in bytes, and s denotes the size of a node in a T-tree, CSS-tree or enhanced B+-tree measured in cache-lines. We choose typical values as follows: $R = K = P = 4$ bytes, $n = 10^7$, $h = 1.2$, $c = 64$ bytes and $s = 1$.

Time Analysis: To make the analysis easy, in this section we assume that R, P and K are the same. Thus we have a single parameter m, which is the number of slots per node. The size of a node in cache-lines is given by $s = \frac{mK}{c}$.

The first table in Table 1 shows the branching factor, number of levels, comparisons per internal node and comparisons per leaf node for each method. B+-trees have a smaller branching factor than CSS-trees since they need to store child pointers explicitly. The total cost of each searching method has three parts, namely the comparison cost, the cost of moving across levels and the cache miss cost. We show the three costs for each method in the second table in Table 1. We use D to denote the cost of dereferencing a pointer, A_b, A_{fcss}, A_{lcss} to denote the cost of computing the child address for binary search, full CSS-tree and level CSS-tree respectively. First of all, the comparison cost is more or less the same for all the methods. Full CSS-trees have slightly more comparisons than level CSS-trees as we described earlier. Some of the methods find the child node by following pointers and others by arithmetic calculations. The relative cost depends on computation complexity and the hardware. For example, while A_b could be smaller than D, A_{fcss} is likely to be more expensive than D. Nevertheless, methods with a higher branching factor have fewer levels and thus usually have a lower cost of moving across levels. But that doesn't mean the larger the branching factor the better. Too large a node size will increase the cache miss cost, which is probably the most important factor since each cache miss can

Method	Space (indirect)	Typical Value	Space (direct)	Typical Value	RID-Ordered Access
Binary search	0	0 MB	0	0 MB	Y
Full CSS-trees	$\frac{nK^2}{sc}$	2.5 MB	$\frac{nK^2}{sc}$	2.5 MB	Y
Level CSS-trees	$\frac{nK^2}{sc-K}$	2.7 MB	$\frac{nK^2}{sc-K}$	2.7 MB	Y
enhanced B+-trees	$\frac{nK(P+K)}{sc-P-K}$	5.7 MB	$\frac{nK(P+K)}{sc-P-K}$	5.7 MB	Y
Hash table	$(h-1)nR$	8 MB	hnR	48 MB	N
T-trees	$\frac{2nP(K+R)}{sc-2P}$	11.4 MB	$\frac{2nP(K+R)}{sc-2P}+nR$	51.4 MB	Y

Table 2: Space analysis

be an order of magnitude more expensive than a unit computation.

We assume a cold start in the cache. If the node size is smaller than the cache line size, each level has one cache miss. When the node size is larger than the cache line size, we estimate the number of cache misses per node to be $(\log_2 s)+\frac{1}{s}=(\log_2 \frac{mK}{c})+\frac{c}{mK}$ (the total number of cache misses for all the keys adds up to $s*(\log_2 s)+1$ and we divide that by s assuming each cache-line is equally likely to be chosen). The results are summarized in the last two columns of the table. For most reasonable configurations, the number of cache misses is minimized when the node size is the same as cache line size. In the third column, we can see that binary search and T-trees always have a number of misses that are independent of m. B+-trees and CSS-trees have only a fraction of the cache misses of binary search. The fractions for CSS-trees are even smaller than that of B+-trees. So CSS-trees have the lowest values for the cache related component of the cost. As we can see, as m gets larger, the number of cache misses for all the methods approaches $\log_2 n$ (essentially all methods degrade to binary search). There is a tradeoff between full CSS-trees and level CSS-trees. While the latter has slightly more cache misses, it also performs fewer comparisons. It's hard to compare the moving cost of the two since A_{lcss} is cheaper than A_{fcss}. We will show an experimental comparison in Section 6.

To summarize, we expect CSS-trees to perform significantly better than binary search and T-trees in searching, and also to outperform B+-trees. If a bunch of searches are performed in sequence, the top level nodes will stay in the cache. Since CSS-trees have fewer levels than all the other methods, it will also gain the most benefit from a warm cache.

Space Analysis: Table 2 lists the space requirements of the various algorithms. The column "Space (indirect)" describes the space required by the algorithms if the structure being indexed is a collection of record-identifiers that can be rearranged if necessary. In other words, it is acceptable for a method to store record-identifiers internally within its structure, as opposed to leaving the list of record-identifiers untouched as a contiguous list. In this column, we do not count the space used by the record-identifiers themselves since all methods share this space requirement.

The column "Space (direct)" describes the space required by the algorithms if the structure being indexed is a collection of records that *cannot* be rearranged. In other words, it is not acceptable for a method to store the records internally within its structure. In this column, we count the space used by record-identifiers for T-trees and hash tables since record-identifiers would not be necessary with other methods.

All methods other than hash tables support access in RID-order. The formula for Level CSS-trees assumes that $\frac{sc}{K}$ is a power or 2.

6 Experimental Results

We perform an experimental comparison of the algorithms on two modern platforms. We analyze the wall-clock time taken to perform a large number of random successful lookups to the index. 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. 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 chained bucket hashing, array binary search, interpolation search [Pet57], T-tree, enhanced B+-tree, full CSS-tree and level CSS-tree in C++. We chose to implement all the existing methods ourselves because we are considering them in the context of main memory OLAP environment. Existing implementations won't be optimized for space allocation and cache related issues such as alignment. As a result, we believe our implementation will run faster. Since cache optimization can be sensitive to compilers [SKN94], we also chose two different compilers: one is Sun's native compiler CC and the other is GNU's g++. We used the highest optimization level of both compilers. However, the graphs for different compilers look very similar, so we only report the results for CC. All the keys are integers and are chosen randomly between 0 and 1 million. Each key takes four bytes. The lookup keys are generated in advance to prevent the key generating time from affecting our measurements. We performed 100,000 searches on randomly chosen matching keys. We repeated each test five times and report the minimal time to find the first matching key.

Algorithm Implementation Details: We tried our best to optimize all the searching methods. For methods that can have different node sizes, we implemented specialized versions for selected node size. We allocate a large chunk of memory at the beginning and create all the nodes from that pool to save allocation time. We use logical shifts in place of multiplication and division whenever possible. For T-trees, B+-trees and CSS-trees, we unfold the binary search loop in each internal node by hardcoding all the *if-else* tests to reduce the amount of overhead. The search within a leaf node is also hardcoded. Additionally, once the searching range is small enough, we simply perform the equality test sequentially on each key. This gives us better performance when there are less than 5 keys in the range. Code specialization is important. When our code was more "generic" (including a binary search loop for each node), we found the performance to be 20% to 45% worse than the specialized code.

The sorted array is aligned properly according to the cache line size. For T-trees, B+-trees and CSS-trees, all the tree nodes are allocated at once and the starting addresses are also aligned properly.

We implemented enhanced B+-trees. They use all the slots in each node.[1] Each node is designed to have a prespecified size and all the nodes are aligned properly. We also forced each key and child pointer to be adjacent to each other physically.

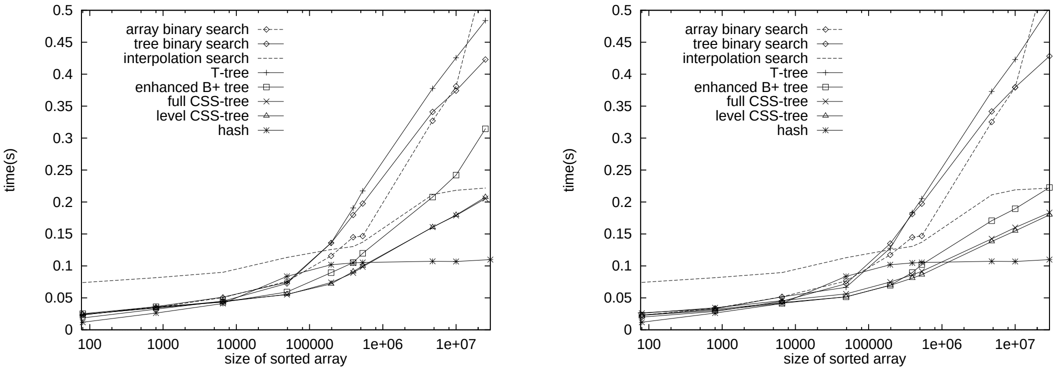
We avoid storing the parent pointer in each node of a T-tree since it's not necessary for searching. We implemented the improved version of T-Trees [LC86b], which is a little bit better than the basic version. For each T-tree node, we store the two child pointers adjacent to the smallest key so that they will be brought into the same cache line together. (Most of the time, the improved version checks only the smallest key in each node.)

For the chained bucket hashing, we followed the techniques used in [GBC98] by using the cache line size as the bucket size. Besides keys, each bucket also contains a counter indicating the number of occupied slots in the bucket and the pointer to the next bucket. Our hash function simply uses the low order bits of the key and thus is cheap to compute.

[1]Since there is always one more pointer than keys, for nodes with an even number of slots, we leave one slot empty.

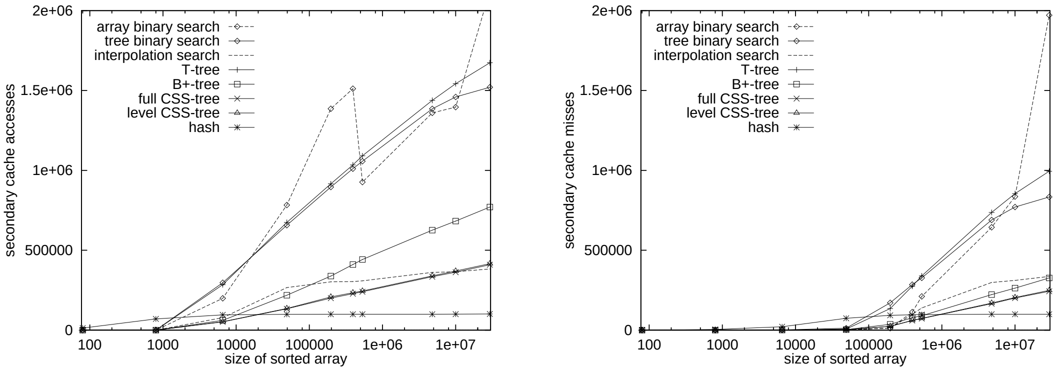
Results: In the first experiment, we test how long it takes to build a CSS-tree. Both building time curves are linear in the size of the sorted array. Level CSS-trees are cheaper to build than full CSS-trees because they don't need to traverse each subtree to find the largest key. We are not claiming that CSS-trees can be built much faster than other index structures. Instead, we want to show that since the absolute rebuilding time is small (less than a second for 25 million keys), we can afford to rebuild CSS-trees periodically. The graph is omitted due to space limitations.

We then measure the searching time for all the algorithms. We first fix the node size and vary the size of the sorted array. We choose the node size to be each of the cache line size of the two levels of cache in the Ultra Sparc (32 bytes and 64 bytes). Figure 5 shows the result on the Ultra Sparc. First of all, when all the data can fit in cache, there is hardly any difference among all the algorithms (except for interpolation search). As the data size increases, we can see that our cache conscious CSS-trees perform the best among all the methods except for hashing. T-tree and binary search are the worst and run more than twice as slow as CSS-trees. The B+-tree curve falls in the middle. Although these searching methods all have to do the same number of key comparisons, they differ on how many of those comparisons cause a cache miss. T-tree search and binary search essentially have one cache miss for each comparison. For CSS-trees, all comparisons within a node are performed with only one cache miss. B+-trees also have one cache miss per node. But since a B+-tree stores half as many keys per node as CSS-trees, it has more levels than a CSS-tree. Although it's hard to discern visually, the level CSS-tree performs slightly better than the full CSS-tree. Across all of our tests we observed that level CSS-trees were up to 8% faster than full CSS-trees. The performance of interpolation search depends on how well the data fits a linear distribution. Although not shown here, we also did some tests on non-uniform data and interpolation search performs even worse than binary search. So in practice, we would not recommend using interpolation search. For B+-trees and the two CSS-trees, the numbers in Figure 5(b) are smaller than that in Figure 5(a). The reason is that the miss penalty for the second level of cache is larger than that of the on-chip cache. B+-trees get more benefit than CSS-trees from having a larger node size. This is because B+-trees always hold half the number of keys per node as CSS-trees and thus its benefit from avoiding extra cache misses is more significant. Although the T-tree has many keys per node, it doesn't benefit from having a larger node size. In this experiment, we chose the hash table directory size to be 4 million. Hashing uses



(a) 8 integers per node (b) 16 integers per node

Figure 5: Varying array size, Ultra Sparc II



(a) Total number of key accesses (b) Cache misses in secondary level cache

Figure 6: Ultra Sparc II, 16 integers per node

only one third of the time of CSS-trees. But we have to keep in mind that it's using 20 times as much space.

To compare the number of cache misses, we ran all the searching methods through a cache simulator.[2] Figure 6(a) and 6(b) show the total number of key accesses and the total number of key misses in the secondary level cache in a configuration that matches the Sun Ultra Sparc. As the data set gets larger, a larger proportion of the key accesses are cache misses. B+-trees have approximately 50% more cache misses than CSS-trees. Let's assume that each RAM access takes 180ns.[3] Then enhanced B+-trees take 0.06 seconds to load data from RAM to cache (when there are 25 million keys in the sorted array) and this is about 30% of the total amount of time taken (0.22 seconds). Out of the 0.04 seconds time difference between B+-trees and CSS-trees, about 0.014 seconds is caused by the difference in cache miss penalties. Thus the cache simulation verifies that 30% of the observed performance gap is due to the cache miss behavior. As the speed gap between CPU and RAM widens, this ratio will get higher and the difference between the performance of each method will be even more significant.

The results on the Pentium PC are very similar and are omitted here. In [LC86a, LC86b], the authors reported that T-trees perform better than binary search and B+-trees. Our results show the contrary conclusion. The explanation is that the CPU speed has improved by two orders of magnitude relative to memory latency during the past thirteen years [CLH98].

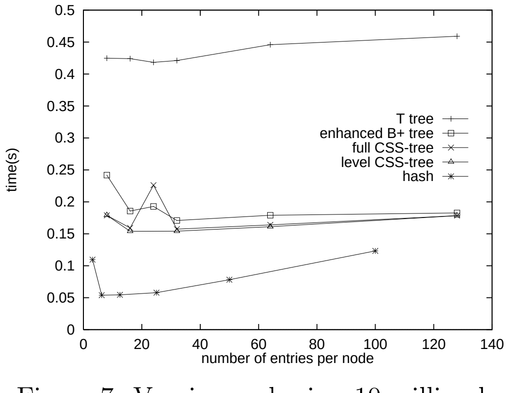


Figure 7: Varying node size, 10 million keys

In our next experiment, we fix the size of the sorted array and vary the node size of T-trees, B+-

[2] We implemented the simulator ourselves and instrumented our code (using `#ifdef SIMULATOR` macros) to log all memory accesses.

[3] We assume the memory has a peak bandwidth of 350MB/s [Inc99]. 64/350M=180ns

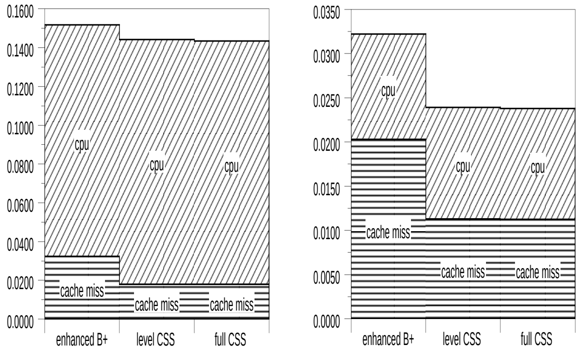
trees, full CSS-trees and level CSS-trees. Figure 7 shows the result on the Ultra Sparc. For both CSS-trees, the lowest point is 16 integers per node, which corresponds to the cache line size of the machine. B+-trees have a minimum at 32 integers per node. Although this doesn't quite fit in the time analysis in Section 5, the difference between 16-integers and 32-integers per node is not very significant. There is a bump for full CSS-trees and B+-trees when each node has 24 integers. One reason is that the node size is not a multiple of the cache line size. Thus nodes are not properly aligned with cache line and cause more cache misses. The bumps in full CSS-trees are more dramatic. This is because that the arithmetic computation for the child node is more expensive for $m = 24$ since division and multiplication must be used to compute child nodes instead of logical shifts. We also tested the optimal hash directory size of chained bucket hashing. Each point corresponds to a directory size from 2^{23} to 2^{18} with the leftmost point having the largest directory size. When the directory size has been increased to 2^{23}, the system starts to run out of memory and thus the searching time goes up. T-trees perform much worse on all the node sizes.

7 Space/Time Tradeoffs

An indexing method is measured by the pair (S, T) of space required for the whole index structure (S), and time taken for a single lookup (T). Figure 2 shows the space requirement and searching time for each method.[4] Each point for T-trees, B+-trees and CSS-trees corresponds to a specific node size. Since keeping an ordered RID list is usually necessary, we calculate the space requirement using the formula for "direct" space listed in Table 2. The stepped line basically tells us how to find the optimal searching time for a given amount of space. All the methods on the upper-right side of the line are dominated by some methods on the line. T-trees and B+-trees are both dominated by CSS-trees. Methods on the line have tradeoffs between space and time. On the bottom, we have binary search on the sorted array, which doesn't require any extra space, but uses three times as much time as CSS-trees and seven times as much as hashing. If we invest a little bit of extra space, we can use CSS-trees, which reduce the searching time to one third of binary search. To make another factor of two improvement, we have to pay 20 times as much space as CSS-trees to be able to use hashing. When space is limited, CSS-trees balance the space and time.

We show the breakdown of the cost of enhanced B+-trees and CSS-trees in Figure 8(a). For the time being, the cache miss cost corresponds to a relatively small fraction of the total cost. Given the trend shown in Figure 1, in five years, we expect the CPU to be 10 times faster while the memory speed will be only 1.6 times faster. So the breakdown at that time will probably look like that in Figure 8(b). The cache miss cost will be much more significant. Figure 9 shows the projected space/time tradeoff between B+-trees and CSS-trees in five years. As we can see, the optimal CSS-trees are almost 30% faster than the optimal enhanced B+-trees. In the limit, if cache miss cost dominates the total cost, CSS-trees can be up to 50% faster than enhanced B+-trees. Also, the optimal node size in five years is different from what it is now.

[4] We present the numbers from searching a sorted array of 5 million elements on an Ultra Sparc II.



(a) now (b) in five years

Figure 8: Time breakdown (16 integers per node)

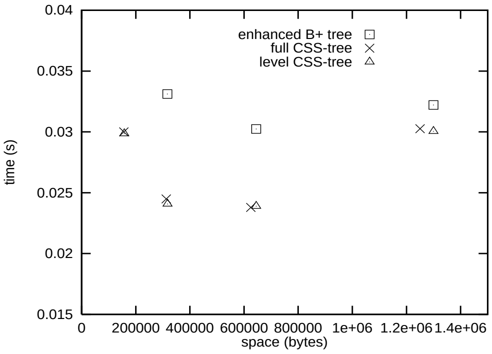


Figure 9: Space/time Tradeoffs (in five years)

8 Conclusion

We studied the cache behavior of various in-memory indexes in an OLAP environment and showed that cache conscious methods such as CSS-trees can improve searching performance by making good use of cache lines. As the gap between CPU and memory speed is widening, we expect the improvement that can be achieved by exploiting the cache will be even more significant in the future. Cache conscious searching behavior is just one step towards efficiently utilizing the cache in database systems. We aim to study the effect of cache behavior on other database operations in the future.

Acknowledgment: We thank John Grogan for the implementation of our cache simulator.

References

[AHK85] Arthur C. Ammann, et al. Design of a memory resident DBMS. In *Proceedings of the IEEE COMPCOM Conference*, pages 54–57, 1985.

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

[CLH98] Trishul M. Chilimbi, James R. Larus, and Mark D. Hill. Improving pointer-based codes through cache-conscious data placement. Technical report 98, University of Wisconsin-Madison, Computer Science Department, 1998.

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

[CS83] F. Cesarini, et al. An algorithm to construct a compact B-tree in case of ordered keys. *Information Processing Letters*, 17(1):1612–1630, 1983.

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

[Fre95] Clark D. French. "One size fits all" database architectures do not work for DDS. In *Proceedings of the ACM SIGMOD Conference*, page 449–450, 1995.

[Fre97] Clark D. French. Teaching an OLTP database kernel advanced data warehousing techniques. In *Proc. IEEE Int'l Conf. on Data Eng.*, 1997.

[GBC98] Goetz Graefe, et al. Hash joins and hash teams in Microsoft SQL server. In *Proceedings of the 24th VLDB Conference*, pages 86–97, 1998.

[GMS86] Hector Garcia-Molina, et al. High perfromance transaction processing with memory resident data. In *Proceedings of the International Workshop on High Performance Transaction Systems*, 1986.

[GR93] Jim Gray, et al. *Transaction processing:concepts and techniques*. The Morgan Kaufmann Publishers, San Francisco, CA, USA, 1993.

[Hag86] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computing*, C-35:839–842, 1986.

[Inc99] Sun Microsystems Inc. Datasheet on memory (available from http://www.sun.com/ microelectronics/datasheets/sme1040/04.html as of feb. 15, 1999). 1999.

[JLRS94] H.V. Jagadish, et al. Dali: A high performance main memory storage manager. In *Proceedings of the 20th VLDB Conference*, 1994.

[JTR87] Wiebren De Jonge, et al. Two access methods using compact binary trees. *IEEE Transactions on Software Engineering*, 13(7):799–810, 1987.

[Knu73] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, USA, 1973.

[LC86a] Tobin J. Lehman and Michael J. Carey. Query processing in main memory database management systems. In *Proceedings of the ACM SIGMOD Conference*, pages 239–250, 1986.

[LC86b] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th VLDB Conference*, pages 294–303, 1986.

[LC87] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high performance memory-resident database system. In *Proceedings of the ACM SIGMOD Conference*, pages 104–117, 1987.

[LL96] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1, 1996.

[LL97] Anthony LaMarca, et al. The influence of caches on the performance of sorting. In *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[LN88] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *International Symposium on Databases in Parallel and Distributed Systems*, pages 177–189, 1988.

[LSC92] Tobin J. Lehman, et al. An evaluation of starburst's memory resident storage component. *IEEE Transactions on knowledge and data enginnering*, 4(6):555–566, 1992.

[NBC+94] Chris Nyberg, et al. Alphasort: a RISC machine sort. In *Proceedings of the ACM SIGMOD Conference*, pages 233–242, May 1994.

[Pet57] W. W. Peterson. In *IBM J. Res. & Devel.*, No. 1, pages 131–132, 1957.

[SAC+79] Patricia G. Selinger, et al. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference*, pages 23–34, 1979.

[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.

[Sof97] TimesTen Performance Software. Main-memory data management technical white paper (available from http://www.timesten.com). 1997.

[Syb97] Sybase Corporation. *Sybase IQ 11.2.1*, 1997.

[WK90] Kyu-Young Whang and Ravi Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems*, 15(1):67–95, 1990.

[WL91] Michael E. Wolf, et al. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, 1991.