Main-Memory Index Structures with Fixed-Size Partial Keys

Philip Bohannon
Lucent Technologies
Bell Laboratories
Murray Hill, NJ
bohannon@bell-labs.com

Peter McIlroy
Lucent Technologies
Murray Hill, NJ
pmcilroy@lucent.com

Rajeev Rastogi
Lucent Technologies
Bell Laboratories
Murray Hill, NJ
rastogi@bell-labs.com

ABSTRACT

The performance of main-memory index structures is increasingly determined by the number of CPU cache misses incurred when traversing the index. When keys are stored indirectly, as is standard in main-memory databases, the cost of key retrieval in terms of cache misses can dominate the cost of an index traversal. Yet it is inefficient in both time and space to store even moderate sized keys directly in index nodes. In this paper, we investigate the performance of tree structures suitable for OLTP workloads in the face of expensive cache misses and non-trivial key sizes. We propose two index structures, pkT-trees and pkB-trees, which significantly reduce cache misses by storing *partial-key information* in the index. We show that a small, fixed amount of key information allows most cache misses to be avoided, allowing for a simple node structure and efficient implementation. Finally, we study the performance and cache behavior of partial-key trees by comparing them with other main-memory tree structures for a wide variety of key sizes and key value distributions.

Keywords

cache coherence, B-tree, T-tree, key compression, main-memory indices

1. INTRODUCTION

Following recent dramatic reductions, random access memory (RAM) is competitive in price with the disk storage of a few years ago. With multi-gigabyte main memories easily affordable and expandable (on 64-bit architectures), applications with as much as 1 or 2 GB of data in main memory can be built with relatively inexpensive systems, and moderate growth in space requirements need not be a concern. For these reasons, and spurred by the stringent performance demands of advanced business, networking and internet applications, a number of main-memory database and main-memory database cache products have appeared in the market [2, 22, 29]. These products essentially fulfill the expectations of research on main-memory databases of the last fifteen years (see, for example, [9, 12, 15, 16]), by providing an approximate order-of-magnitude performance improvement for simple database applications, when compared to disk databases with data *fully resident* in main memory [2, 29].

Adapting main-memory database algorithms to become "cache-conscious," that is, to perform well on multi-level main-memory storage hierarchies, has recently received attention in the database literature [5, 24, 25]. As mentioned out in these papers and in related work (see, for example, [6]), commonly used processors can now execute dozens of instructions in the time taken for a read from main memory (a "cache miss"). For instance, memory access time on a 450 MHz Sun ULTRA 60 is more than 50 times slower than the time to access data resident in the on-chip cache. Further, the disparity between processor speed and memory latency is only expected to grow since CPU speeds have been increasing at a much faster rate (60% per year) than memory speeds (10% per year) [6, 24]. Consequently, main-memory index structures should be designed to minimize cache misses during index traversal, while keeping CPU costs and space overhead low. Intuitively, cache-miss costs are minimized with small node sizes and high branching factors. For example, [6] found that optimal node sizes for their B-tree implementation was slightly larger than 1 cache block (so that the average number of keys present in a node would fill a cache block). Low CPU costs for index traversal are important since cache misses cost no more than a few-dozen instructions. In this setting, key comparison costs are an important component of CPU cost, especially for multi-part or variable-length keys. Additionally, space overhead is important since the cost of RAM is approximately $1/MB, or about 50 times as expensive as disk storage. As a result, the amount of main memory available to the index may be limited by cost factors, leading to constraints on index size. The space usage depends on the space used to represent keys in index nodes, the space used for pointers, and the average occupancy of nodes in the tree.

For main-memory OLTP environments which include a mix of read and update operations, the T-tree[1] and the B^+-tree are two index structures which have been studied previously in the literature [17, 24]. All main-memory database products of which we are aware [2, 29], implement the T-tree index structure proposed by Lehman and Carey [17]. However, in [24], the authors found that due to the higher cost of cache misses on modern hardware, B^+-trees performed better in experiments conducted with integer keys. While the assumption of integer keys may be valid in an OLAP environment assuming suitable pre-processing, a general purpose database must handle complex keys – multiple parts, null values,

[1] The T-tree is similar to a binary tree with multiple keys (instead of one) stored in each node.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA
Copyright 2001 ACM 1-58113-332-4/01/05 ...$5.00.

variable-length fields, country-specific sort values, etc. Further, key size and key storage strategy directly affect the branching factor for B- or B^+-trees. Since branching factors are small already for node sizes based on cache blocks, the height of the tree can vary substantially as key size changes. Thus an initial motivation for our research was the further examination of T-tree and B-tree performance in a main-memory OLTP environment, in order to consider a variety of key storage schemes and key sizes.

In [9, 17], the authors suggest avoiding the key size problem by replacing the key value in the index with a pointer to the data and reconstructing the key as needed during index traversal. This *indirect* key-storage approach has the advantage of optimizing storage by eliminating duplication of key values in the index, improving the branching factor of nodes and simplifying search by avoiding the complexity of storing long or variable-length keys in index nodes. However, this approach must be re-examined due to the additional cache misses caused by retrieval of indirect keys. A second approach to dealing with large, complex keys is to use key compression to allow more keys to fit in cache blocks. The key-compression approach has the benefit that the entire key value can be constructed without accessing data records or dereferencing pointers. However, typical compression schemes such as employed in prefix B-trees [4] have the disadvantage that the compressed keys are variable-sized, leading to undesirable space management overheads in a small, main-memory index node. Further, depending on the distribution of key values, prefix-compressed keys may still be fairly long resulting in low branching factors and deeper trees.

In this paper, we propose the *partial-key* approach, which uses fixed-size parts of keys and information about key differences to minimize the number of cache misses and the cost of performing compares during a tree traversal, while keeping a simple node structure and incurring minimal space overhead. A key is represented in a partial-key tree by a pointer to the data record containing the key value for the key, and a partial key. For a given key in the index, which we refer to as the *index key* for the purposes of discussion, the partial key consists of (1) the offset of the first bit at which the index key differs from its *base* key, and (2) l bits of the index key value following that offset (l is an input parameter). Intuitively, the base key for a given index key is the most recent key encountered during the search prior to comparing with the index key. The partial-key approach relies on being able to resolve most comparisons between the search key and an index key using the partial-key information for the index key. If the comparison cannot be resolved, the pointer to the data record is dereferenced to obtain the full index key value.

Using the idea of partial keys, we develop the *pkT-tree* and the *pkB-tree*, variants of the T-tree and B-tree, respectively. We describe search algorithms for these partial-key trees as well as strategies for maintaining the partial-key information in the presence of updates. Finally, we conduct an extensive performance study of the pkT-tree and pkB-tree structures, comparing them to standard T-trees and B-trees with both direct and indirect key storage schemes. In our experiments, we consider a wide range of parameter settings for key size and key value distribution (entropy). We also study the sensitivity of our partial-key algorithms to l, the number of key value bits stored in the partial key. Our performance results, given in detail in Section 5.3, indicate that:

- Of the indexing schemes studied, partial-key trees minimize cache misses for all key sizes.
- Due to lower CPU costs, B-trees with direct key storage are faster than partial-key trees for small key sizes, but slower for larger key sizes.
- Partial-key schemes have good space utilization, only slightly worse than T-trees with indirect key storage, and much better than direct key storage schemes.
- A small, fixed value for l (the amount of partial key information) avoids most indirect key accesses for a wide variety of key lengths and entropies.

In summary, partial-key trees incur few cache misses, impose minimal space overheads and reduce the cost of key comparisons without introducing variable-length structures into the node, thus enabling larger keys to be handled with much of the efficiency of smaller keys. Further, we expect the relative performance of partial-key trees to improve over time with the increasing cost of cache misses.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. In sections 3 and 4, we introduce partial-key comparisons and apply them to search in pkT- and pkB-trees. In Section 5, we present the results of our performance study. Finally, in Section 6, we present our conclusions and issues to be addressed in future work.

2. RELATED WORK

An early study of index structures for main-memory databases was undertaken in [17]. The authors proposed the T-tree index and, in order to optimize storage space, advocated storing pointers to data records instead of key values in the index. However, this design choice can result in a large number of cache misses since each pointer dereference to access the key value during a key comparison could potentially lead to a cache miss. Since at the time of this early work on main-memory databases, there was little difference between the cost of a cache-hit and that of a cache-miss, not much attention was paid to minimizing cache block misses. While most work on cache-conscious data structures outside of the database community has focused on optimizing scientific workloads, cache-conscious behavior was studied in [6] for "pointer-based" structures including search trees. However, this work focused on actions which can be taken without programmer cooperation, rather than explicitly designed data structures.

More recently, Rao and Ross propose two new main-memory indexing techniques, *Cache-Sensitive Search Trees* (CSS-tree) [24] and *Cache-Sensitive B^++-Trees* [25]. Designed for a read-intensive OLAP environment, the CSS-tree is essentially a very compact and space-efficient B^+-tree. CSS-tree nodes are fully packed with keys and laid out contiguously, level by level, in main memory. Thus, children of a node can be easily located by performing simple arithmetic, and explicit pointers to child nodes are no longer needed. Further, in the absence of updates, key values can be mapped to integers such that the mapping preserves the ordering between key values. Thus, each key value in a CSS-tree is a compact integer, which is stored in the node itself, eliminating pointer dereferences. In summary, the CSS-tree incurs very little storage space overhead and exhibits extremely good cache behavior. The CSB^+-tree adapts these ideas to an index structure which supports efficient update (for the CSS-tree, the authors recommend rebuilding from scratch after a batch of updates). This structure stores groups of sibling nodes adjacent in memory, reducing the number of pointers stored in the parent node without incurring additional cache misses. However, this work continues to assume integer keys. To this extent, the performance improvements of CSB^+-trees or CSS-trees and partial-key trees are likely to be orthogonal, since the former focuses on reducing pointer overhead and improving space utilization while the latter focuses on reducing key-size and comparison cost.

Our partial-key techniques borrow from earlier work on key compression [4, 11]. However, there are differences, which we discuss below. Partial-key trees are most similar to Bit Trees that were introduced in [11]. Bit Trees extend B$^+$-trees by storing partial keys instead of full key values for (only) those keys contained in leaf nodes. The partial key in a Bit Tree consists of only the offset of the difference bit relative to the previous key in the node. The authors describes several properties of searches using only the offset of difference bits, and in particular show the somewhat surprising result that the precise position of a search key in a leaf node can be determined by performing *exactly one* pointer dereference to retrieve an indirect key. Other than a focus on main-memory rather than disk, our partial-key trees differ from Bit Trees in the following respects: (1) partial keys are stored in both internal nodes and leaf nodes, (2) partial keys contain l bits of the key value following the difference bit in addition to the difference bit offset, and (3) searching for a key in a node in a partial-key tree requires at most one pointer to be dereferenced, and frequently requires no pointer dereferences, due to the l bits of additional information and our novel search algorithms.

Prefix B$^+$-trees, proposed in [4], employ key compression to improve the storage space characteristics and the branching factor of B$^+$-trees. Suppose p is the common prefix for keys in the subtree rooted at node N. For a key $k = p \cdot s$ in node N, the common prefix p can be computed during tree traversal and only the suffix s of the key value is stored in N. Further, when keys move out of a leaf node due to a split, only the *separator*, or the shortest portion of the key needed to distinguish values in the splitting nodes, is moved. Partial-key trees differ from prefix B$^+$-trees in the following respects: (1) while prefix trees factor out the portion common to all keys in a node, partial-key trees factor out information in common between pairs of adjacent keys within the node, typically a longer prefix than is common to the whole node, (2) while in prefix B$^+$-trees, the entire suffix of the separator is stored, in partial-key trees, only the first l bits of the suffix is stored – thus, partial-key trees may lose key value information while the prefix B$^+$-tree does not, (3) in the prefix B$^+$-tree no pointer dereferences are needed. In contrast, in a partial-key tree, pointer dereferences must be performed when the comparison cannot be resolved using the partial-key information, and (4) the partial keys stored in a prefix B$^+$tree are variable sized and this complicates implementation. Further, in some cases the separator may not even fit in a 64-byte cache line, causing index nodes to span multiple cache blocks and reducing the branching factor. Thus partial key trees trade off the guarantee of no indirect key references of partial-key trees for a low probability of indirect key dereferences, in exchange for simple node structures and more strongly bounded tree heights. This is reasonable since the cost of a cache-miss is orders of magnitude lower than the cost of a random disk access.

Ronstrom in his thesis [28] describes the HTTP-tree, a variation of prefix B$^+$ trees in which further compression is performed within a node by storing keys relative to the previous key, factoring out common suffixes, etc. Nodes are also clustered on pages to facilitate distribution. However, during searches the full key is reconstructed in order to perform comparisons, and compressed key sizes are variable. Other lossless compression schemes primarily for numeric attributes have recently been proposed in the database literature [13, 23]. Goldstein, Ramakrishnan and Shaft [13] propose a page level algorithm for compressing tables. For each numeric attribute, its minimum value occurring in tuples in the page is stored separately once for the entire page. Further, instead of storing the original value for the attribute in a tuple, the difference between the original value and the minimum is stored in the tuple.

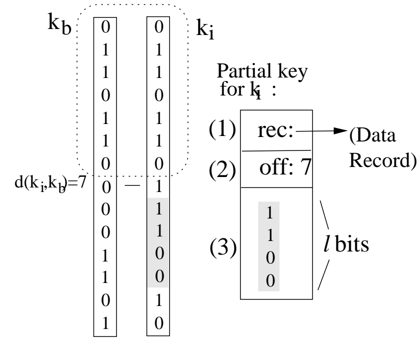

Figure 1: A Partial Key

Thus, since storing the difference consumes fewer bits, the storage space overhead of the table is reduced. Tuple Differential Coding (TDC) [23] is a compression method that also achieves space savings by storing differences instead of actual values for attributes. However, for each attribute value in a tuple, the stored difference is relative to the attribute value in the preceding tuple.

3. PARTIAL KEY SEARCH

In this section, we describe the *partial key* approach and algorithms for performing compares and searches in the presence of partial keys. We assume that keys are represented as fixed-length bit strings (though this is not required in general). Further, the bits are numbered in order of *decreasing* significance, beginning with bit 0 (the most significant bit).

3.1 Partial Keys – An Overview

Consider the order in which index keys are visited and compared with the search key during a traversal of a T-tree or B-tree index. We observe that for both structures, the key visited so far which is *closest* in value to the search key is either the most recent key to compare less than the search key or or the most recent key to compare greater. It is easy to see that the most recent key to compare less than the search key shares more *initial bits* than any other key which compared less than the search key during the search. Similarly for the most recent key which compared greater, thus the observation follows. In fact, very few initial bits may be shared between the most recent with the search key when, for example, the keys are on either side of a large power of two, but this event is rare. In the following discussion, we refer to the current key being visited as the *index* key (as opposed to search key), and the previous key visited as the *base* key. In our partial-key schemes, each key is represented in the index by three items: (1) a pointer to the record containing the key, (2) the offset of the first bit at which the index key differs from the base key, and (3) the first l bits of the index key following this offset. We illustrate the construction of a partial key in Figure 1. In this figure, and in others below, the index key or search key is generally referred to as k_i or k_j, and the base key as k_b.

One approach to using this partial key information would mirror the use of prefixes in a prefix B-tree. In this approach, the search code would maintain the known prefix of the index key as it traversed the tree, concatenating appropriate portions of partial keys as they are encountered. If the known portion is sufficient to resolve a comparison with the search key, then a cache miss is avoided. However, it turns out that constructing this prefix is not necessary, and in fact comparisons can often be resolved by noting the offset at which the search key differed from the base key, and comparing

that to the offset stored in the partial key for the index key. This observation ensures that most comparisons are performed with small, fixed-length portions of the key. Precisely how these comparisons are performed is the topic of the next section.

3.2 Partial-Key Comparisons

In this section we discuss the properties of difference bits more formally, and present a theorem that bears directly on comparisons in partial-key trees.

Let $d(k_i, k_j)$ be the offset of the most significant (thus lowest) bit of difference between keys k_i and k_j. Also, let $c(k_i, k_j)$ be the result, LT, GT or EQ, of the comparison between keys k_i and k_j depending on whether k_i is $<$, $>$, or $= k_j$. The partial-key approach is based on the observation that for an index key k_j and its base key k_b, noting $d(k_j, k_b)$ in the partial key for key k_j will frequently allow full comparisons of k_j with a search key k_i to be avoided during index retrieval. In particular, if the two keys k_i and k_j compare in the same way (LT, for example) to the base key, k_b, and if $d(k_i, k_b)$ and $d(k_j, k_b)$ are known, then it is possible to determine how k_i compares to k_j as well as $d(k_i, k_j)$ without additional reference to the keys unless $d(k_i, k_b) = d(k_j, k_b)$.

THEOREM 3.1. *Given k_i, k_j and k_b as above, if $c(k_i, k_b) = c(k_j, k_b)$ and $d(k_i, k_b) \neq d(k_j, k_b)$, then*

$$d(k_i, k_j) = \min(d(k_i, k_b), d(k_j, k_b))$$

and

$$c(k_i, k_j) = \begin{cases} c(k_b, k_i) & \textit{if}\ d(k_i, k_b) > d(k_j, k_b) \\ c(k_i, k_b) & \textit{if}\ d(k_i, k_b) < d(k_j, k_b) \end{cases}$$

PROOF. Assume without loss of generality that $k_i, k_j < k_b$. Suppose that $d(k_i, k_b) > d(k_j, k_b)$. Since $k_j < k_b$, then at the first bit of difference, $d(k_j, k_b)$, k_j's bit must be 0 and k_b's bit must be 1. It follows from $d(k_i, k_b) > d(k_j, k_b)$ that the bits of k_i agree with the bits of k_b for all bits up to and including $d(k_j, k_b)$, so the corresponding bit of k_i is also 1. Thus, $k_i > k_j$ and $d(k_i, k_j) = d(k_j, k_b)$. On the other hand, suppose that $d(k_i, k_b) < d(k_j, k_b)$. Thus, since $k_i < k_b$, the bit at position $d(k_i, k_b)$ must be 0 in k_i and 1 in k_b. Further, since $d(k_i, k_b) < d(k_j, k_b)$, this bit must be the same in both k_j and k_b, that is, 1. Further, the bit sequences preceding this bit in both k_i and k_j must be identical. Thus, it follows that $k_i < k_j$ and $d(k_i, k_j) = d(k_i, k_b)$. The other cases for $k_i, k_j > k_b$ follow from symmetry. □

The key ideas of Theorem 3.1 are illustrated in Figure 2(a). Here, keys k_i and k_j are both less than the base key k_b, and $d(k_i, k_b) = 9$ is greater than $d(k_j, k_b) = 5$. Thus, as shown in the figure, $d(k_i, k_j) = 5$ and $k_i > k_j$ because the first 5 bits of k_i, k_j and k_b match, but on the 6th bit, k_i and k_b are both 1 while k_j is 0.

Theorem 3.1 can be used to compute, *for most cases*, the result of the comparison between a search key k_i and the index key k_j. This is because the partial key for an index key k_j stores the difference bit offset $d(k_j, k_b)$ with respect to its base key k_b that was encountered previously in the search. Further, since an attempt is made by the search algorithm to compare k_i with k_j, it must be the case that $c(k_j, k_b) = c(k_i, k_b)$. Also, $d(k_i, k_b)$ and $c(k_i, k_b)$ are available due to the previous comparison between the search key and the base key. Thus, for the case when $d(k_i, k_b) \neq d(k_j, k_b)$, Theorem 3.1 can be used to infer $d(k_i, k_j)$ and $c(k_i, k_j)$, and these in turn can be propagated to the next index key comparison (for which k_j is the base key).

The only case not handled by Theorem 3.1 occurs when $d(k_i, k_b) = d(k_j, k_b)$. In this case, the only inference one can make is that k_i and k_j are identical on the first $d(k_i, k_b) + 1$ bits. However, one cannot determine how keys k_i and k_j compare. An example of this is illustrated in Figure 2(b). In both of the cases shown in the figure, $k_i, k_j < k_b$ and $d(k_i, k_b) = d(k_j, k_b)$. However, in one, $k_i < k_j$ and in the other $k_j > k_i$. When the difference bits are equal, the l bits of the key value stored for the index key are compared with the corresponding bits in the search key. If these bits are equal, retrieval of the indirectly stored key is required. Note that, as shown in Figure 1, the difference bit itself in not included in the l bits stored with a partial key. Since both k_i and k_j differ from k_b in the value of this bit, the corresponding bits in k_i and k_j must be identical.

```
procedure COMPAREPARTKEY(searchKey, indKey, comp, offset)
begin
1.  if (indKey.pkOffset < offset)
2.      if (comp = LT)
3.          comp := GT
4.      else
5.          comp := LT
6.      offset := indKey.pkOffset
7.  else if (indKey.pkOffset = offset)
8.      if (comp = LT)
9.          partKey := searchKey[0:indKey.pkOffset-1]
                 ·0·indKey.partKey;
10.     else
11.         partKey := searchKey[0:indKey.pkOffset-1]
                 ·1·indKey.partKey;
12.     comp, offset := compare(searchKey,
13.         partKey);
14.     if (offset > indKey.pkOffset + indKey.pkLength)
15.         comp := EQ;
16. return [comp, offset];
end
```

Figure 3: COMPAREPARTKEY**: Comparison using Partial Keys.**

Procedure COMPAREPARTKEY in Figure 3 utilizes Theorem 3.1 to compute the result of a comparison between a search key and an index key containing partial-key information. The partial-key information consists of three fields pkOffset, pkLength and partKey which are described in Table 1. The input parameters comp and offset to COMPAREPARTKEY are the result of the comparison and the difference bit location of the search key with respect to the base key. Steps 1–6 of the procedure are a straightforward application of Theorem 3.1. In case the difference bit locations for the search key and index key are equal, the keys must be identical until the difference bit. Further, the difference bit itself in both keys must be either 0 or 1 depending on whether the keys are less than or greater than the base key. In Step 12, function COMPARE is invoked to compute the comparison of the l bits following the difference bit in both keys. Function COMPARE(k1, k2) returns a pair of values comp, offset with the following semantics. The value comp is one of EQ, LT or GT depending on whether the bit sequence k1 is equal to, less than or greater than the bit sequence k2 when the two sequences are compared bit by bit. The return value offset is the location of the most significant bit in which the two keys differ. Thus, in steps 13–14, since partKey may not represent the entire index key, if all bits in partKey agree with the corresponding bits in searchKey, the comparison between the search key and index key cannot be resolved and the procedure returns EQ. In this case, the semantics of the returned offset is simply that the two keys agree on the first offset-1 bits.

Note that, in Step 12, function COMPARE only needs to consider bits starting from bit offset indKey.offset in the two keys (since the corresponding bits preceding this point are identical for both

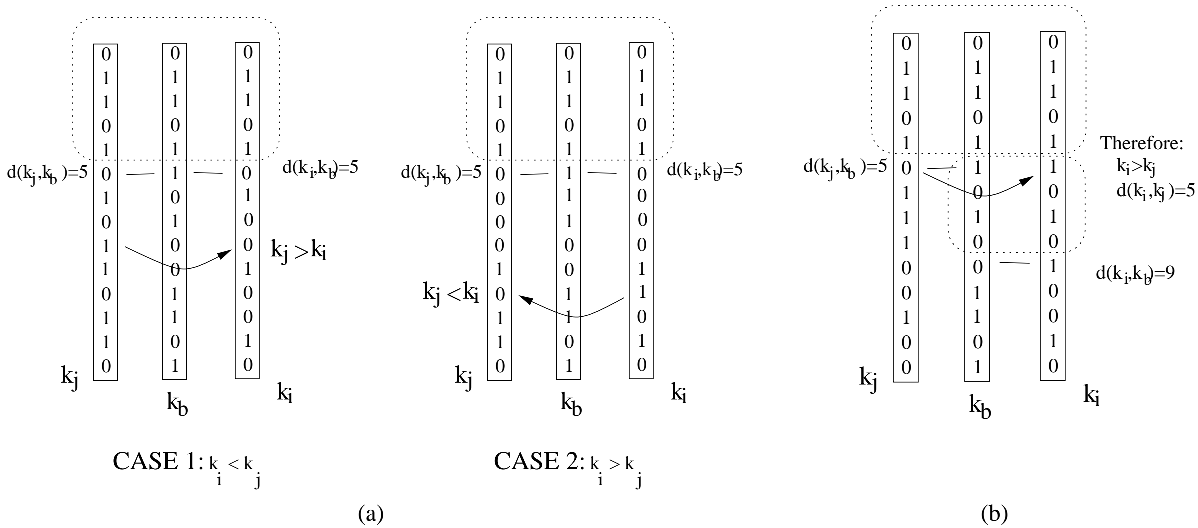

Figure 2: Examples of Comparisons between Keys k_i and k_j that Can and Cannot be Resolved

Table 1: Partial-Key Notation

Symbol	Description
l	Maximum number of bits from key value stored in partial key
k_i.pkOffset	Offset of difference bit of key k_i and its base key
k_i.partKey	Upto l bits following location k_i.pkOffset in key k_i
k_i.pkLength	Number of partial-key bits stored with key k_i
$k_i[l : m]$	Bit sequence consisting of bits between offsets l and m in key k_i
$k_1 \cdot k_2$	Concatenation of bit sequences k_1 and k_2
N.numKeys	Number of keys in node N
N.key[i]	i^{th} key in node N
N.ptr[i]	i^{th} pointer in node N

keys). Further, in most cases, procedure COMPAREPARTKEY performs only one integer comparison involving difference bit offsets; however, additional expense may be incurred since the bit of offset must be computed in anticipation of the next comparison. While a greater cost than simple integer compares, it compares well to comparisons of larger keys, as shown in Section 5.

The partial-key scheme can be adapted to multi-segment keys, even if some segments are of arbitrary length. The idea is to treat the pkOffset as a two digit number, where the first digit indicates the key segment and the second indicates an offset within that segment. The partKey field may be limited to bits from a single segment, or at the cost of more complexity, span segments.

3.3 Partial-Key Nodes

The COMPAREPARTKEY procedure described in the previous section is the basic building block for performing retrievals in pkT-trees and pkB-trees. Before we present the complete algorithm for searching in a tree, we present below a *linear encoding* scheme for computing the partial keys for an array of keys in an index node N. We also present a linear search algorithm for finding a search key in the node if it is present, and if it is not, the pair of adjacent keys in the node between which the search key lies.

Linear Encoding of Partial Keys. The base key for each key in N is simply the key immediately preceding it in N. For the first key N.key[0], the base key is a key in an ancestor of N in the tree that is compared with the search key during the tree traversal before node N is visited. Thus, the base key for the first key depends on the tree structure and is different for the pkT-tree and pkB-tree. We discuss this further in the following section).

Simple Linear Search Algorithm. When searching for a key in the index, the node N is visited after the search key is compared with the base key for N.key[0]. Let comp and offset denote the result of the comparison and the offset of the difference bit between the search key and the base key. Then, in order to locate the position of the search key in N, procedure COMPAREPARTKEY (see Figure 3) can be used to compute the comparison and the difference bit offset between the search key and N.key[0]. In case the comparison cannot be resolved using the partial-key information for N.key[0] (that is, COMPAREPARTKEY returns EQ), then N.key[0] is dereferenced and the search key is compared with the full key corresponding to N.key[0]. The result of the comparison and difference bit offset is then used to compare the search key with N.key[1], and so on. The above steps of comparing with the search key are repeated for successive keys in N with the comparison and difference bit offset for the previous key – until a key that is greater than or equal to the search key is found.

However, this naive linear search strategy may perform unnecessary key dereferences. The following example illustrates this.

EXAMPLE 3.2. *Consider the node N in Figure 4. The difference bit for each key (with respect to the previous key) is marked by an arrow and $l = 1$ bits following the difference bit are stored in the partial key for each key. Let the base key for the first key in N*

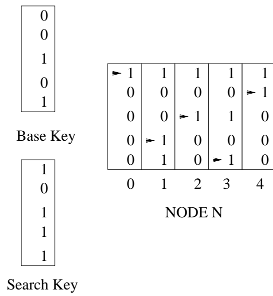

Figure 4: Linear Searching for Key in Node

be 00101 and let the search key be 10111.

After the search key is compared with the base key, the comparison and difference bit offset are GT and 0, respectively. Invoking COMPAREPARTKEY *with the search key,* N.key[0]*, GT and 0 returns [EQ, 2] since* N.key[0].pkOffset $= 0$ *and the search key matches with* N.key[0] *on the first two bits. Thus,* N.key[0] *would be dereferenced, and* comp *and* pkOffset *after the comparison with* N.key[0] *is GT and 2, respectively. Next the search key is compared with* N.key[1] *by invoking* COMPAREPARTKEY*. Since* N.key[1].pkOffset $= 3$ *which is greater than 2, the difference bit offset for the search key and* N.key[0]*,* COMPAREPARTKEY *returns [GT, 2], and* N.key[1] *is not dereferenced. The next invocation of* COMPAREPARTKEY *with* N.key[2] *returns [GT, 3] since* N.key[2].pkOffset $= 2$ *and the bit sequence 1010 for the constructed key for* N.key[2] *is smaller than 1011, the corresponding bits of the search key. After returning [GT, 3] for* N.key[3] *(since* N.key[3].pkOffset $= 4$ *is greater than 3), procedure* COMPAREPARTKEY *moves to key 4 and returns [LT, 1] for* N.key[4] *since it finds that* N.key[4].pkOffset $= 1$ *is less than 3, the offset returned for* N.key[3]*. Thus, the simple linear search algorithm stops at* N.key[4] *and the position of the search key in N is determined using only one key dereference, that of* N.key[0].

However, the position of the search key can also be determined without dereferencing any keys, including N.key[0]*. The reason for this is that* COMPAREPARTKEY *returns [EQ 2] when it is invoked with* N.key[0]*. Thus, at this point, we know that the first two bits of* N.key[0] *are 10 since the first two bits of* N.key[0] *agree with those of the search key. Since* N.key[1].pkOffset $= 3$*, we also can conclude that the first two bits of* N.key[1] *agree with those of* N.key[0] *are thus 10. Since* N.key[2].pkOffset $= 2$ *and* N.key[2] $>$ N.key[1]*, it must be the case that the third bit of* N.key[2] *is 1 (and the third bit of* N.key[1] *must have been 0). Further, the fourth bit of* N.key[2] *can be obtained from its partial key and thus we can conclude that the first four bits of* N.key[2] *are 1010. Since the search key is 10111, the comparison between the search key and* N.key[2] *can be resolved and is [GT, 3]. Subsequent comparisons can then be carried out as described earlier to conclude that the search key lies in between* N.key[3] *and* N.key[4]*. Thus, the position of the search key can be determined without dereferencing a single key.* ∎

Linear Search Algorithm Requiring at most One Key Dereference. Procedure FINDNODE, shown in Figure 5, avoids the unnecessary dereferences made by the simple linear search algorithm. When comparing the search key with an index key in node N, in case procedure COMPAREPARTKEY returns EQ, that is, the comparison between a search key and index key cannot be resolved, FINDNODE does not immediately dereference the index key. Instead, it exploits the semantics of [EQ, offset] returned by COMPAREPARTKEY (which is that the search key and the index key agree on the first offset-1 bits) to try and resolve comparisons with subsequent keys. This was illustrated earlier in Example 3.2, where [EQ, 2], the result of the comparison with N.key[0], was useful in resolving the comparison operation with key N.key[2]. In fact, procedure COMPAREPARTKEY as already stated correctly handles the value of EQ as an input parameter when called from FINDNODE, , and an informal proof of this fact can be found in Appendix A.

```
procedure FINDNODE(N, searchKey, offset)
begin
1.  high := N.numKeys;
2.  low := -1;
3.  cur_off := offset;
4.  cur_cmp := GT;
5.  cur := low +1;
6.  while (cur < high){
7.      cur_cmp, cur_off :=
8.          COMPAREPARTKEY(searchKey, N.key[cur],
                cur_cmp, cur_off)
9.      if (cur_cmp = LT)
10.         high := cur;
11.         break;
12.     else if (cur_cmp = GT)
13.         low := cur;
14.         offset := cur_off;
15.     cur++;
16. }
17. if (high - low > 1)
18.     low, high, offset := FINDBITTREE(N, searchKey, low, high)
19.         /* cache miss */
20. return [low, high, offset];
end
```

Figure 5: FINDNODE**: Linear Searching for a Key in a Node Using Partial Keys.**

Procedure FINDNODE accepts as input parameters node N in which to perform the linear search and the difference bit offset between the search key and the base key for N.key[0]. It assumes that both the search key and N.key[0] are greater than the base key with respect to which N.key[0]'s partial key is computed. Similar to the simple linear search algorithm described earlier, it compares the search key with successive index keys in the node until an index key larger than the search key is found. Procedure COMPAREPARTKEY is used to perform every comparison and the results of the previous comparison (stored in variables cur_cmp and cur_off) are passed as input parameters to it. Unlike the simple linear search scheme, an index key is not immediately dereferenced if the precise result of the comparison between the key and the search key cannot be computed. Instead, variables low and high are used to keep track of the positions of index keys in N that the search key is definitely known to be greater than and less than, respectively.

At the end of a sweep of keys in node N, if high - low is greater than 1, then it implies that the precise position of the search key in N is ambiguous. In this case, procedure FINDBITTREE is used to locate the exact position of the search key between low and high – it returns the consecutive keys between which the search key lies and the difference bit offset of the search key with respect to the lower key. Procedure FINDBITTREE employs the search algorithm for Bit Trees described in [11] and requires exactly one key to be dereferenced. In a nutshell, the algorithm performs a sequential scan of keys in N between offsets low and high. It maintains a variable pos, which is initially set to low. For each key examined, if for the difference bit offset in its partial key, the bit value in the search key is 1, then variable pos is set to the position of the key in

N. On the other hand, if the bit value in the search key is 0, then keys for which the difference bit offset is greater than the current key's, are skipped and the next key examined is the key whose difference bit offset is less than that of the current key. Once all the keys have been examined, N.key[pos] is dereferenced and is compared with the search key. If comp, offset is the pair returned by compare(searchKey, N.key[pos]), then FINDBITTREE takes one of the following actions:

1. comp = EQ: Return [pos, pos, offset].
2. comp = GT(LT): Suppose that high is the position of the first key to the right(left) of pos for which the difference bit offset is less than that of N.key[pos]. Return [high-1, high, offset].

The correctness of FINDBITTREE for Case 2 above is due to the following property of pos: Between pos and a key whose difference bit offset is equal to that of pos, there is a key whose difference bit offset is less than pos's. We refer the reader to [11] for details.

The variable offset returned by FINDNODE is the difference bit offset between the search key and N.key[low - 1]. In case low = -1, that is, the search key is less than N.key[0], then offset is the difference bit offset between the search key and the base key for N.key[0]. Finally, if the search key is contained in N, then the procedure returns the position of the index key that equals the search key. Revisiting Example 3.2, FINDNODE determines the position of the search key in node N without requiring any keys to be dereferenced. Successive invocations of procedure COMPAREPARTKEY for the sequence of keys in N return [EQ, 2], [EQ, 2], [GT, 3], [GT, 3] and [LT, 1].

Maintaining Partial-Key Information in the Presence of Updates. With the linear encoding strategy, maintaining the partial-key information is quite straightforward. Insertion of a new key in the node requires the partial keys of the inserted key and the key following it to be recomputed, while deletion requires only the partial key for the key following the deleted key to be recomputed.

4. PARTIAL-KEY TREES

Building on the partial-key comparison and single-node partial-key search algorithms presented in the previous section, we now discuss how partial keys can improve performance in main-memory index structures, by reducing the L2 cache miss rate. In particular, we present partial-key variants of the T-tree [17] and B-tree [3] index structures suitable for use in main-memory. The pkT-ttree and pkB-tree, as we refer to them, extend their main-memory counterparts by representing keys by their partial-key information as described in Section 3. The linear encoding scheme described in the previous section is used to compute partial keys for the index keys in a node. With linear encoding at the node level, we thus only need to specify the base key with respect to which the first key in each node is encoded since the base key for every other key in the node is simply the key preceding it.

4.1 pkT-tree

The T-tree is a balanced binary tree with multiple keys stored in each node. The leftmost and the rightmost key value in a node define the range of key values contained in the node. Balancing is handled as for AVL trees [1]. We refer the reader to [17] and [26] for additional information about T-trees, including details of update strategies and concurrency control. The pkT-tree is similar to the T-tree except that in addition to a pointer to the data record containing the full key value, each index key entry also contains partial-key information. In the following, N.ptr[0] and N.ptr[1] denote pointers to the left and right children of node N.

Storing Partial-Key Information. For the first key in each node N, the base key with respect to which the partial key is computed is the first key in the parent node. This is because, as described below, the leftmost key in the parent node is the key with which the search key is compared before node N is visited. Figure 6(a) depicts an example pkT-tree – in the figure, the solid arrows denote the base keys for index keys while the dashed arrows represent pointers to child nodes.

```
procedure FINDTTREE(searchKey, T)
begin
1.  N := root of tree T;
2.  laN := nil;
3.  offset := 0;
4.  comp := GT;
5.  while (N != nil) {
6.      comp, offset := COMPAREPARTKEY(searchKey,
                            N.key[0], comp, offset);
7.      if (comp = EQ)
8.          dereference N.key[0] /* cache miss */
9.          comp, offset := compare(searchKey,N.key[0]);
10.     if (comp = EQ)
11.         return [N, 0, 0, offset]
12.     else (comp = LT)
13.         N := N.ptr[0];
14.     else
15.         laN := N;
16.         laNOffset := offset;
17.         N := N.ptr[1];
18. }
19. return [laN, FINDNODE(laN - laN[0], searchKey, laNOffset)];
end
```

Figure 7: FINDTTREE**: Searching for a Key in T-tree Using Partial Keys.**

Searching for Key. Searching for a key value in the pkT-tree is relatively straightforward and is performed as described in procedure FINDTTREE (see Figure 7). Procedure FINDTTREE includes an optimization from [17] which requires that at each node, the search key be compared to only the leftmost key value the node (Step 6). The variables comp and offset keep track of the results of the most recent comparison of the search key with the leftmost key in the parent node, and are passed as parameters to COMPAREPARTKEY. In case COMPAREPARTKEY cannot resolve the comparison, the leftmost key in the current node is dereferenced (Step 8). If the search key is found to be less than this key, the search proceeds with the left subtree. If it is found to be greater, the search proceeds with the right subtree, with the current node noted in variable laN (Step 15). The significance of laN is that when the search reaches the bottom of the tree, the search key, if present in the tree, is in the node stored in laN and is greater than laN[0]. Procedure FINDNODE can thus be employed in order to determine the position of the search key in node laN. Since FINDNODE requires the leftmost key to be greater than the base key for it, laN[0] is deleted from laN before passing it as an input parameter to FINDNODE.

Maintaining Partial-Key Information in the Presence of Updates. Inserts and deletes of keys into the pkT-tree can cause rotations, movement of keys between nodes and insertions/deletions

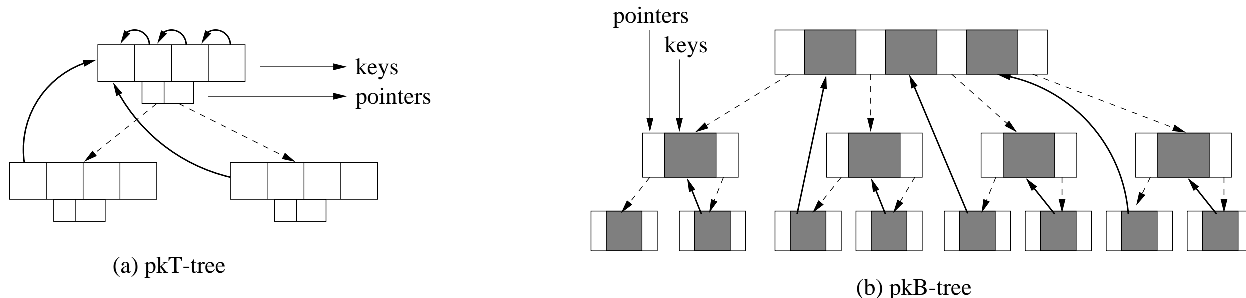

Figure 6: pkT-trees and pkB-trees

of keys from nodes. The partial-key information for these cases is updated as follows.

- In the case of a rotation, the parent of a node involved in the rotation may change. Thus, the partial-key information for the leftmost key in the node is recomputed with respect to the new parent.
- If the leftmost key in a node changes, the partial-key information is recomputed for the leftmost keys in the node and its two children.
- In case the key to the left of a key in a node changes (due to a key being inserted or deleted), the partial-key information for the key is recomputed relative to the new preceding key.

4.2 pkB-Tree

The pkB-tree is identical to a B-tree except for the structure of index keys. Each index key consists of a pointer to the data record for the key and partial-key information. Leaf nodes contain only index keys, while internal nodes also contain $N.\mathsf{numKey}+1$ pointers to index nodes – the subtree pointed to by $N.\mathsf{ptr}[i]$ contains keys between $N.\mathsf{key}[i-1]$ and $N.\mathsf{key}[i]$.

Storing Partial-Key Information. The base key for the leftmost key in N is the largest key contained in an ancestor of N that is less than the leftmost key. Thus, if N' is the node such that $N'.\mathsf{ptr}[i]$, $i \neq 0$, points to N or one of its parents, and for all N'' in the path from N' to N, $N''.\mathsf{ptr}[0]$ points to N or one of its parents, then the base key relative to which $N.\mathsf{key}[0]$ is encoded is $N'.\mathsf{key}[i-1]$. This is illustrated in Figure 6(b), where the solid arrows denote the base keys for index keys of the pkB-tree and the dashed arrows represent pointers to child nodes.

Searching for Key. Procedure FINDBTREE, in Figure 8, contains the code for searching for a key in a pkB-tree. Beginning with the root node, for each node, procedure FINDNODE is invoked to determine the child node to be visited next during the search. The variable offset stores the offset of the difference bit between the search key and the base key for $N.\mathsf{key}[0]$ (that is, the largest key in an ancestor of N that is also less than $N.\mathsf{key}[0]$). This is because FINDNODE simply returns the offset input to it if the search key is less than $N.\mathsf{key}[0]$.

Maintaining Partial-Key Information in the Presence of Updates. An insert operation causes a key to be inserted in a leaf node of the pkB-tree. If the key is inserted at the leftmost position in the leaf, then its partial key needs to computed relative to the largest key that is less than it in its ancestor. On the other hand, if there are keys to the left of it in the node, then its partial key is computed easily relative to its preceding key and the partial key of the next key is computed with respect to it. In case node N is split, the splitting key from N is inserted into its parent. Splits can thus be handled by simply updating the partial-key information in the parent similar to the key insertion case.

Key deletion from a pkB-tree is somewhat more complicated. In case the leftmost key in a leaf is deleted, the partial-key information for it needs to be recomputed from its base key in its ancestor. Deletion of a non-leftmost key in the leaf simply requires the partial key for the key following it to be recomputed. Finally, deletion of a key $N.\mathsf{key}[i]$ from an internal node N of the pkB-tree causes it to be replaced with the smallest key in the subtree pointed to by $N.\mathsf{ptr}[i+1]$. Suppose N' is the node containing this key. Then for every node N'' between N and N'', $N''[0]$'s partial key is recomputed with this smallest key (that replaces the deleted key $N.\mathsf{key}[i]$) as the base key.

```
procedure FINDBTREE(searchKey, T)
begin
1.  N := root of tree T;
2.  pN := nil;
3.  offset := 0;
4.  while (N != nil) {
5.      pN := N;
6.      low, high, offset := FINDNODE(N, searchKey, offset);
7.      if (low = high)
8.          return [N, low, high]
9.      N := N.ptr.[high]
10. }
11. return [pN, low, high];
end
```

Figure 8: FINDBTREE: Searching for a Key in B-tree Using Partial Keys.

5. PERFORMANCE

The goal of our performance study was to compare the lookup performance of T-trees, B-trees, pkT-trees and pkB-trees in a main-memory setting. Our particular goals were as follows: (1) Study performance over a wide range of key sizes and key value distributions. (2) Evaluate the impact of changing the amount of partial key information used for pkT- and pkB-trees. (3) Evaluate space usage and the space-time tradeoff. In subsequent sections we describe our hardware platforms, the design of our experiments and present selected results.

5.1 The Memory Hierarchy

The latencies observed during memory references depend primarily on whether the data is present in cache and whether the vir-

System	CPU	L1 (data)			L2 (data)			DRAM
	Cycle Time	Size	Block	Latency	Size	Block	Latency	L2 Miss Latency
Sun ULTRA 30	3.7ns	16K	64	6ns	2M	64	33ns	266ns
Sun ULTRA 60	2.2ns	16K	64	4ns	4M	64	22ns	208ns
Pentium III	1.7ns	16K	32	5ns	512K	32	40ns	142ns
Pentium IIIE	1.4ns	16K	32	4ns	256K	32	10ns	113ns

Table 2: Latency of Cache vs. Memory.

tual address is in the *Translation Lookaside Buffer* (TLB).[2]

A modern main-memory architecture typically includes two levels of cache, a small, fast, on-CPU L1 cache and a larger, off-CPU,[3] and therefore slower L2 cache. Typical parameters for cache and memory speed are shown in Table 2 (see [19, 20, 7]). The latency information is generated with version 1.9 of *lmbench* [18] on locally available processors, and is intended to give the reader a feel for current cache parameters, not as a comparison of these systems.

Another component of the memory hierarchy is the TLB, which caches translations between virtual and physical addresses. While TLB-misses are shown in [5] to have a significant effect on performance, we do not focus on TLB issues in this paper. One justification for this approach is the fact that almost all modern TLBs are capable of using "superpages" [14], essentially allowing single TLB entries to point to much larger regions. While posing difficulties for operating system implementors [27], this facility may effectively remove the TLB miss issue for main-memory databases by allowing the entire database to effectively share one or two TLB entries. While we do not focus on TLB effects, they were apparent during our experimentation in the form of better performance when index nodes span multiple cache lines (these results are not shown due to limited space). Determining the effect of superpages on the TLB costs of main-memory data structures remains future work.

5.2 Experimental Design

We implemented T-trees and B-trees for direct and indirect storage of keys. We also implemented pkT-trees and pkB-trees, and varied the size of the partial keys stored in the node. Our T-tree algorithms are essentially those of Lehman and Carey, with the optimization of performing only a single key-comparison at any given level. In the direct key and partial-key variants, we stored entire/partial key values only for the *leftmost key* in each node, used during the initial traversal. While the T-tree code was adapted from a system with additional support for concurrency control [26], next-key locking [21] and iterator-based scans, these features were not exercised in our tests.

For the partial-key trees, we implemented two schemes for storing offsets, *bit-wise* and *byte-wise*. The bitwise scheme was used for the description in Section 3, since in this scheme the concepts can be more clearly articulated. However, it may be more convenient in an implementation to store difference information at a larger *granularity*. In particular, we consider the byte granularity. Clearly, all the results of Sections 3 and 4 hold when byte offsets differ, since the bit offset will also differ in the same manner. However, when byte offsets are equal, it may still be the case that the bit offsets would differ. In this case, one simply stores all the bits at which the difference could occur, in other words, the entire byte. Thus, if the offsets compare equal, the keys will be disambiguated by the first byte. Storing offsets at a larger granularity trades off distinguishing power for coding simplicity. With bit offsets one always stores the precise l bits most capable of distinguishing the keys; otherwise, only some of those bits are stored.

Keys. We model keys as unique, fixed-length sequences of unsigned bytes. Key comparisons are performed byte-wise in the context of a separate function call. Indirect keys are stored in separate L2 cache lines since they are typically retrieved from data records.

It is intuitive that partial keys would be sensitive to the distribution of keys, and in particular to the *entropy* of the keys. Since in our tests we generate bytes of the key independently, the entropy for each byte depends only on the number of symbols from which each byte is selected. Specifically, when each byte is selected uniformly from an alphabet of n symbols, each byte contains $\lg n$ bits of *Shannon entropy* [8]. Intuitively, keys with higher entropies will be distinguished earlier during the compare, leading to lower comparison costs. In terms of partial keys, lower entropy leads to larger common prefixes, and a lower chance that two keys will differ within the l bytes of a common prefix.

While, as mentioned in Section 3, partial-key trees may be used with multi-part and variable-length keys, we did not implement these options in our tests. We note that to some degree, not implementing a wider variety of keys (and thus more expensive key comparisons) works against partial-key schemes, since these schemes reduce the impact of key comparison costs in cases when the partial key is sufficient. However, byte-wise comparisons may be somewhat less efficient than, for example, single-instruction integer comparisons. We selected byte-wise comparison as a reasonable model of key comparison expense, and did not attempt to vary this cost as an additional parameter in the current study.

Performance Metrics. We evaluated the various indices based on the following three performance metrics: wall-time, number of L2 cache misses and storage space requirements. The number of L2 misses was measured using special registers available on the UltraSPARC via the PerfMon software [10].

Parameter Settings. Unless otherwise stated, each index node spanned three L2-cache blocks for a total of 192 bytes, each index stored 1M keys, and keys were chosen uniformly and at random, but rejected if they were not unique. Three cache blocks were chosen because that size could handle larger in-node key sizes and, in our experiments (not shown) performed comparably or better than smaller or larger node sizes for all of the studied algorithms. 1M records represented the largest size our machine could easily hold in memory – a relatively large number of records is required to see the effect of L2 cache misses on timing numbers. For most tests, we present results for two choices for the byte entropy for the generated keys: 3.6 bits and 7.8 bits, corresponding to alphabet sizes of 12 and 220, respectively, though in the actual experiments we considered a wide variety of entropies in-between. In most of the experiments, key size was the independent parameter. We fixed the

[2] The physical characteristics of the memory module, especially repeated access to the same page, may also be a factor, but are not considered in this paper.

[3] In the Pentium IIIE, the L2 cache, though relatively small, is on-chip.

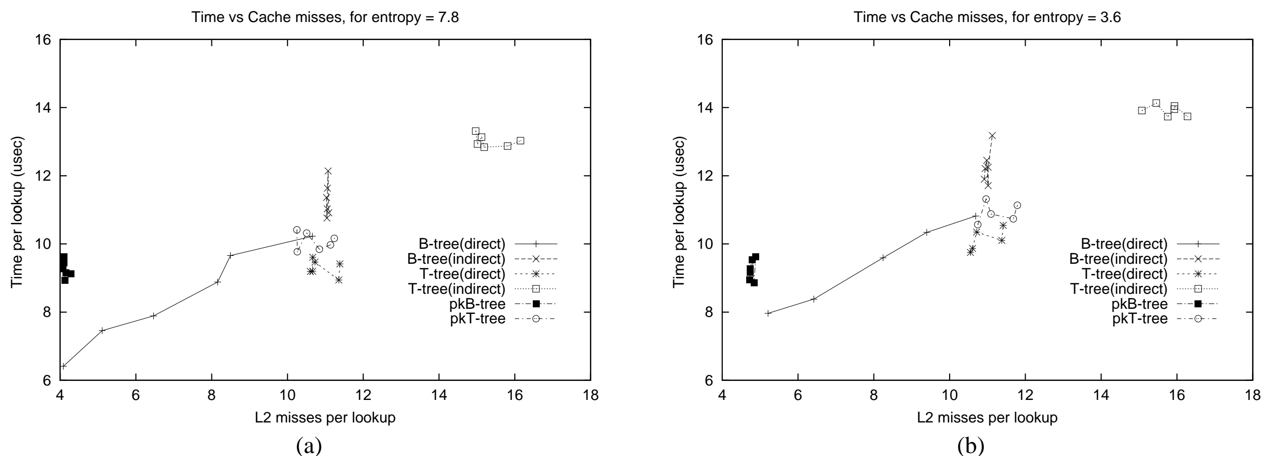

Figure 9: Time and L2 Cache Performance of Various Key Strategies, High and Low Entropy

size l of partial keys stored at 2 bytes and stored offsets in a byte granularity since we found partial-key trees to perform optimally or near-optimally for these choices.

Hardware Environment. Our experiments were conducted on a Sun Ultra 30 workstation with a 296MHZ UltraSPARC II processor and 256 Megabytes of RAM. As shown in Table 2, this machine has a 16K L1 data cache with 32 byte block size and a 2M L2 direct-mapped cache with a 64 byte block size. The latencies shown by *lmbench* [18] are 6ns or 2 cycles for the L1 cache, 33ns or 11 cycles for L2, and 266 ns or 88 cycles for main memory. We implemented the index structures using Sun's C++ compiler version 4.2 and optimization level -O3. In our experiments, we ensured that all virtual memory accessed during the runs was resident in RAM.

Experimental Runs. In most cases, a run consisted of 100,000 lookups from a (pregenerated) list of randomly selected keys from the tree. All searches were successful. Each run was repeated 10 times and averaged. We ensured that the overall standard deviation on time was very low (less than 1%). All figures shown in this document are for a tree with 1.5 million elements, the maximum that fit into main memory on the platform.

5.3 Selected Results

Index performance, in a main-memory environment, is dominated by CPU costs of performing key comparisons and cache miss costs. Thus, it is reasonable that B-trees with direct key storage will perform better than partial-key trees for small keys, since space usage is comparable to the space required for the partial key, and our partial-key comparison code is somewhat more expensive than simple byte-wise key compares. However, as keys become longer, B-tree performance can be expected to become worse than partial-key trees due to a lower branching factor and higher key comparison costs at low byte-entropies. In all cases, we expect indirect indexes to perform poorly in comparison with direct indexes of the same data structure, because indirect indexes will require an extra cache misses to perform each comparison. These expectations are confirmed by our experiments.

Figure 9 summarizes the experimental results for all indexing schemes, on data sets with 1.5 million elements. The Y-axis shows the number of L2 cache misses; the X-axis shows the average time of a lookup in microseconds. Plots are parametric in key size, with key sizes 8, 12, 20, 28, and 36 bytes; the high entropy case has an additional point at key size 4. Figure 9(a) shows behavior for low entropy, with entropy per byte of 3.6. Figure 9(b) shows the same experiment, run with entropy 7.8. For a given key size and entropy, down and left defines improved performance. Performance is thus a partial ordering, where one algorithm outperforms another if for all values of key size and entropy value, the algorithm is faster and has fewer cache misses. Using the metrics of cache misses and lookup time, we make the following observations.

- pkB-trees consistently outperform the other algorithms in L2 cache misses.
- Direct B-trees outperform the other algorithms in time for small key sizes, as would hold for integer keys.
- Direct T-trees outperform the other algorithms in time for large key sizes, very slightly outperforming pkB-trees.
- Direct T-trees and indirect B-trees have essentially the same cache performance. This occurs because T-trees suffer about $\lg_2 N$ cache misses due to tree levels, while B-trees suffer about the the same due to key dereferencing.
- Indirect T-trees, perform poorly compared to all other strategies, primarily due to cache misses from both tree levels and key dereferencing.
- For all key sizes, the cache-miss behavior of partial-key trees is as good as that of the corresponding tree structure with direct storage of 4 byte keys.

One of the reasons that the superior cache-miss characteristics of partial-key trees does not always translate into better timing numbers (especially for smaller key sizes) is that other factors like CPU costs for performing key comparisons, etc. are a significant component of the overall performance. However, based on the cache-miss statistics, we expect that the performance of partial-key trees will improve relative to trees with direct key storage as long as processor speeds improve more quickly than main-memory latency.

Choice of l. Larger values of l are necessary when entropy is low, because sufficient entropy must be present in the partial key to have a high probability that it will differ from the corresponding bytes of the search key. (In general, random keys should have length $l \approx 2 \lg_2 N/H$ to ensure that no two keys collide; most keys will be disambiguous at length $\lg_2 N/H$ [8].) One can see that key-wise difference information adapts to low entropy keys: when keys

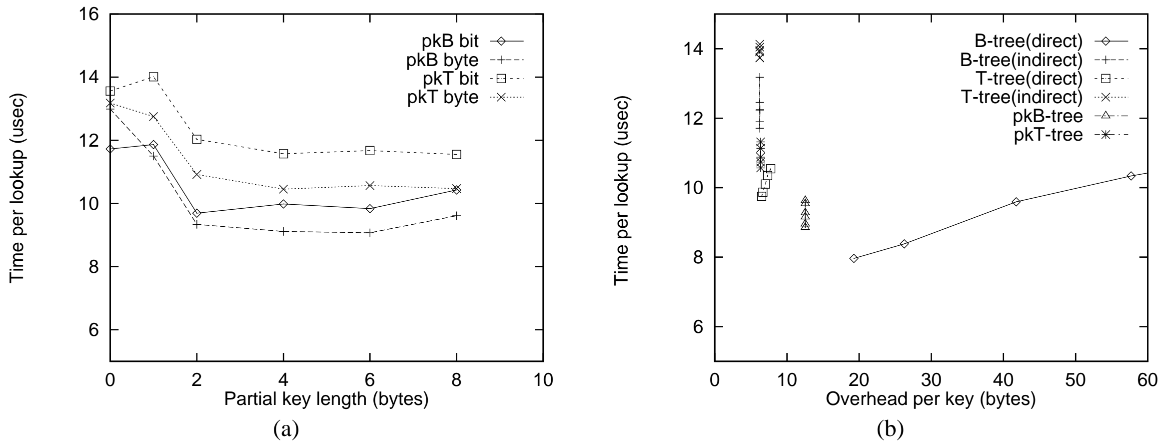

Figure 10: Varying Partial-key Size and Time-Space Tradeoffs

have low entropy, adjacent keys are likely to have larger common prefixes. Further, increasing l adversely affects the branching factor in nodes, thus there is a tradeoff between reducing cache misses by avoiding references to indirect keys and reducing cache misses with bushier, and thus shallower, trees. We investigated these issues by running experiments with a wide variety of key entropies and values for l. In this experiment, the keys have relatively low entropy (3.6 bits per byte), but the results are similar over a wide range of entropy values, and from this we expect partial keys to perform well over a wide variety of key distributions. In fact, performance is almost always optimal with small values of l – 2 or 4 bytes – due to the efficacy of storing difference offsets.

Storing zero bytes of key information is a special case which reduces to an algorithm similar to the Bit Tree [11], but generalized to handle internal levels of the tree and incur fewer cache misses. While this option did not perform as well as $l > 0$, our experiments confirm the following intuition – storing differences at the bit level *is* important for $l = 0$ in order to increase distinguishing power.

Space Usage. Space overhead is a critical attribute of a main-memory index, In Figure 10(b), we show the space-time trade-off of different algorithms for a variety of key sizes. In this graph, the X-axis is space and the Y-axis is lookup time (the lower left-hand corner is optimal). The key size parameter varies between 0 to 8 bytes. The space numbers are obtained from the tree built by random insertions of 1M keys. We see from this graph that indirect key storage, while a poor time performer, excels in space. However, schemes with direct key storage trade space for time, with storage overheads that increase significantly with key size. Again, pkT- and pkB-trees provide a nice tradeoff, taking approximately twice the space of indirect storage for all key sizes, but less space than direct-storage B-trees for all key sizes greater than 4.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced two new index structures, pkT- and pkB-trees, designed to optimize the space, time and cache-miss performance of indices in main-memory, OLTP databases. These index structures are based on partial keys, small fixed-size representations of keys which allow index nodes to retain a simple structure, improve their branching factor and speed up key comparisons, yet resolve most key comparisons without reference to indirectly stored keys. In our performance study, we found that partial-key trees perform better then B-Trees (in which keys are stored directly in the node) for keys larger than 12-20 bytes, depending on key distribution. Further, the partial-key trees incur fewer cache misses than B-Trees with all but the smallest key sizes, leading to an expectation that the performance of pkB-Trees relative to B-trees will improve over time as the gap between processor and main memory speeds widen causing the penalty for a cache miss to be severe. Finally, pkB-Trees take up much less space than standard B-trees for all but the smallest trees.

While pkT-trees, and direct T-trees perform well, pkB-trees perform better and are only slightly larger. However, we expect that over time T-trees will be replaced with variations of the B-tree in main-memory databases, because of their dramatically better L2 cache coherence. For optimal performance on all key sizes, our performance results lead one to consider a hybrid approach in which direct storage is used for small, fixed-length keys and partial-key representations are used for larger and variable-length keys.

In future work, we intend to explore other ways in which architectural trends affect performance-critical main-memory DBMS code. One such trend is the increasing availability of instruction-level parallelism, and a related trend is the increasing cost of branch misprediction and other "pipeline bubbles". A second trend is the availability of "superpages" for TLBs which may significantly reduce the TLB cost of in-memory algorithms.

7. REFERENCES

[1] A. Aho, J. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, H. Korth, P. Mcilroy, J. Miller, P.P.S. Narayan, M. Nemeth, R. Rastogi, S. Seshardi, A. Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. Datablitz storage manager: Main memory database performance for critical applications. In *Proceedings of the 1999 ACM SIGMOD/PIDS International Conference on Management of Data*, June 1999.

[3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

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

[5] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the Twenty-Fifth International Conference on Very Large Databases, Edinburgh*, August 1999.

[6] T.M. Chilimbi, J.R. Larus, and M. Hill. Imporving pointer-based codes through cache-conscious data placement. Technical Report 98, University of Wisconsin-Madison, 1998.

[7] Intel Corporation. Pentium III processor for the SC242 at 450 MHz to 800 MHz datasheet. http://developer.intel.com/design/pentiumiii/datashts/244452.htm, 2000.

[8] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley & Sons, New York, 1991.

[9] D.J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 1–8, Boston, Mass., June 1984.

[10] R. Embody and B. Moore. Perfmon user's guide. http://www.cse.msu.edu/ enbody/perfmon.html.

[11] D. Ferguson. Bit-tree a data structure for fast file processing. *Communications of the ACM*, 35(6):114–120, June 1992.

[12] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[13] J. Goldstein, R. Ramakrishnan, and U. Shaft. "Compressing Relations and Indexes". In *Proceedings of the International Conference on Data Engineering*, Orlando, Florida, 1998.

[14] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4), July 1998.

[15] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main-memory storage manager. In *Proc. of the Int'l Conf. on Very Large Databases*, 1994.

[16] T. Lehman, E. J. Shekita, and L. Cabrera. An evaluation of Starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, December 1992.

[17] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proc. of the Int'l Conf. on Very Large Databases*, pages 294–303, August 1986.

[18] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In USENIX, editor, *USENIX 1996 Annual Technical Conference, January 22–26, 1996. San Diego, CA*, pages 279–294, Berkeley, CA, USA, January 1996. USENIX.

[19] Sun Microsystems. The Ultra 30 architecture, technical white paper. http://www.sun.com/desktop/products/Ultra30/u30.pdf, 1997.

[20] Sun Microsystems. Ultra 60 workstation datasheet. http://www.sun.com/desktop/products/Ultra60/ultra60_datasheet.html, 1998.

[21] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on Btree indexes. In *IBM Almaden Res.Ctr, Res.R. No.RJ7008, 27pp.*, March 1990.

[22] J.P. Morgenthal. Microsoft COM+ will challenge application server market. Technical Whitepaper: http://www.microsoft.com/Com/wpaper/complus-appserv.asp, 1999.

[23] W.K. Ng and C.V. Ravishankar. Block-oriented compression techniques for large statistical databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):314–328, 1997.

[24] J. Rao and K.A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the Twenty-Fifth International Conference on Very Large Databases, Edinburgh*, August 1999.

[25] J. Rao and K.A. Ross. Making B^+-trees cache conscious in main memory. In *(to be published) Proceedings of ACM-SIGMOD 2000 International Conference on Management of Data*, May 2000.

[26] R. Rastogi, S. Seshadri, P. Bohannon, D. Leinbaugh, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main-memory databases. In *Proceedings of the 23rd Int'l Conference on Very Large Databases*, August 1997.

[27] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187, 1995.

[28] Mikael Ronstrom. *Design and Modelling of a Parallel Data Server for Telecom Applications.* PhD thesis, Linköping University, 1998.

[29] The TimesTen Team. In-memory data management for consumer transactions the timesten approach. In *Proceedings of the 1999 ACM SIGMOD/PIDS International Conference on Management of Data*, June 1999.

APPENDIX

A. CORRECTNESS OF COMPAREPARTKEY FOR EQ

When procedure COMPAREPARTKEY is invoked with comp = EQ, the following conditions hold: (1) indexKey is greater than the base key, and (2) searchKey and the base key agree on the first offset -1 bits. We can show that COMPAREPARTKEY as described in Figure 3 performs the comparison between indexKey and searchKey correctly when invoked with comp = EQ. In order to show this, we need to consider the following three cases:

1. indexKey.pkOffset $>$ offset. In this case, all we can conclude is that the first offset - 1 bits of indexKey match those of the search key. However, there is no way to determine the values for bits offset, . . . , indexKey.pkOffset - 1 of the indexKey or their relationship to the corresponding bits of the search key. The primary reason for this is that we do not know whether the search key is greater than, less than or equal to the base key. Thus, the best COMPAREPARTKEY can do is to return [EQ, offset - 1]. An instance of this scenario in Example 3.2 is when COMPAREPARTKEY is invoked with N.key[1] and [EQ, 2], the result of the previous comparison with N.key[0]. The return value is [EQ, 2].
2. indexKey.pkOffset $<$ offset. In this case, since the search key agrees with the base key on the first offset - 1 bits, the first indexKey.pkOffset bits of the index key and search key must match, while the bit following these bits must be 0 in the search key and 1 in the index key (since we know that the index key is greater than the base key). Thus, the search key must be less than the index key and the offset of the difference bit between the search key and index key must be indexKey.pkOffset.
3. indexKey.pkOffset = offset. In this case, one can conclude that the search key and the index key match on the first offset - 1 bits, and the bit at position offset is 1 in the indexKey (since the index key is greater than the base key). Thus, a comparison can be carried out between the bits starting at position offset in the search key and the corresponding sequence of bits in the index key, that is, 1 followed by the bits in the partial key for indexKey. Thus, in Example 3.2, invoking procedure COMPAREPARTKEY with N.key[2] and [EQ, 2], the result of the previous comparison with N.key[1], results in a return value of [GT, 3]. As a result, the comparison with key N.key[2] can be computed even though the result of the comparison with N.key[1] was ambiguous.