Making T-Trees Cache Conscious on Commodity Microprocessors

IG-HOON LEE1,3, SANG-GOO LEE1 AND JUNHO SHIM2,+
1School of Computer Science and Engineering
Seoul National University
Seoul, 151-742 Korea
E-mail: {ihlee; sglee}@europa.snu.ac.kr
2Department of Computer Science
Sookmyung Women’s University
Seoul, 140-742 Korea
E-mail: jshim@sookmyung.ac.kr

Recent research shows that the database performance can be significantly improved by the effective cache utilization of the conventional microprocessors. Researchers have modified existing index structures into ones optimized for CPU cache performance in main memory database environments. A Cache Sensitive B+-Tree is designed to minimize the impact of cache misses for B+-Trees and it has been known to be more effective than other types of main memory index structure including T-Trees. In this paper, we introduce a Cache Sensitive T-Tree (CST-Tree) and show how T-Trees can also be redesigned to be cache sensitive. We present an experimental performance study which shows that our Cache Sensitive T-Trees can outperform the original T-Trees and Cache Sensitive B+-Trees on commodity microprocessors.

Keywords: main memory, index structure, cache, CST-tree, microprocessor, T-tree

1. INTRODUCTION

The rapid increase in the data volumes for the past few decades has intensified the need for high processing power for database. Researchers have actively sought to design and develop new architectures for improving the performance. Recent research shows that the performance can be significantly improved using effective utilization of architectural features and memory hierarchies used by the conventional processors.

Researchers have paid attention to various aspects of main memory databases. The index structure for main memory is one area in which T-Trees were proposed as a prominent index structure for main memory [1].

Recently, [2, 3] claimed that B-Trees may outperform T-Trees owing to the current speed gap between cache access and main memory access. CPU clock speeds have been increasing at a much faster rate than memory speeds [4-6]. The overall computation time becomes more dependent on first level instruction cache misses (L1) and second level data cache misses (L2) than on disk buffer misses. The total number of memory accesses for T-Trees is higher than the one for B+-Trees, in that T-Trees are designed considering
on random access and pointer operations [3]. In the past we considered the effect of buffer cache misses to develop an efficient disk-based index structure. The same applies to the effect of cache misses.

Albeit cache optimization in main memory systems in principle is similar to main memory optimization in a disk-based system, a significant difference is that the management of the cache is done by the hardware and the database system does not have a direct control to improve the cache hit, memory references satisfied by the cache. This is why the database system needs a built-in cache optimized index structure. A careful design considering the characteristics of cache behavior and cache replacement policy may lead to improvement of cache hits. A most well-known cache optimized index structure for main memory database systems is CSB+-Trees (Cache Sensitive B+-Trees) that is a variant of B+-Trees [3].

Previously, well-known main-memory index structure was T-Tree. T-Tree is still being used at various systems such as Oracle Times Ten [7], MySQL Cluster [8], DataBlitz [9], FastDB [10], MonetDB [11]. Those systems were designed to provide high performance on large databases. Therefore, it is significant to make T-Trees cache conscious so that those systems provide high performance.

A feature in a contemporary CPU architecture comes along with the industry that has launched multi-core CPU microprocessors in the market. It has been about 3 years since the first dual core PC processor was introduced in the market. Experts expect that we will have eight-or 16-core microprocessors in a near feature [12, 13]. The trend concurs in the industry that manufactures processors for workstations and server-levels as well. What it has meant to the software research community is to investigate the performance impact that a multi-core processor may offer, and to change the software architecture to exploit a higher performance benefit of the design of new processor. The database community is one of the early birds which found the trend [12, 14]. Research on cache conscious index structures includes one-dimensional index structures [2, 3] as well as multi-dimensional index structures [15, 16] and cache-conscious concurrency control of main-memory indexes [17].

In this paper, we study how to design the existing T-Trees index structure to better utilize the cache and introduce a new index structure CST-Trees (Cache Sensitive T-Trees). We analyze the complexity of CST-Trees, and provide an experimental study to show how the traditional index structures, recently developed cache conscious versions, and proposed CST-Trees actually perform in modern computer environments. We conduct the experiment to check the performances of T-Trees, B+-Tress, CSB+-Trees, and CST-Trees, on contemporary available computer systems equipped with single-core and multi-core CPUs. The experimental result show that our new cache sensitive T-Trees may outperform the original T-Trees and other existing index structures: CSB+-Trees and B+-Trees. The paper also discusses CST+-Trees which has the merit of CST-Tree and does range search by use of link of data nodes.

The rest of this paper is structured as follows. Section 2 presents the related work. The original T-Trees and our analysis with regard to its cache consciousness are provided. In section 3 we introduce our modified cache-conscious Trees and provide the basic algorithms. In section 4 we present the experimental performance study. In section 5 we discuss about variation of CST-Trees to improve range search performance. And finally, conclusions are drawn in section 6.
2. RELATED WORK

Most widely used tree-based index structures may include AVL-Trees, B+-Trees, and T-Trees [1]. The AVL-Tree is a most classical index structure that was designed for main memory [18]. It is a binary search tree in which each node consists of one key field, two (left and right) pointers, and one control field to hold the balance of its subtree (Fig. 1 (a)). The left or right pointer points the left or right sub-trees of which nodes contain data smaller or larger than its parent node, respectively. The difference in height between the left and right sub-trees should be maintained smaller or equal to one. If an update affects a leaf node and leaves the tree unbalanced, i.e., a control field is larger than |1|, a rotation is performed. There are four different rotations; LL (Fig. 1 (b)), LR (Fig. 1 (c)), RR, and RL. The RR and RL operations are symmetric to LL and LR, respectively.

The major disadvantage of an AVL-Tree is its poor storage utilization. Each tree node holds only one key item, and therefore rotations are frequently performed to balance the tree. T-Trees address this problem [1]. In a T-Tree, a node may contain \( n \) keys (Fig. 1 (d)). Key values of a node are maintained in order. Similar to an AVL-Tree, any key stored within a left and right sub-tree should be smaller or larger than the least and largest data of a node, respectively. The tree is kept balanced by the same rotations as for the AVL-Tree.

B-Trees [18] are designed for disk-based database systems and need few node accesses to search for a data since trees are broad and not deep, i.e., multiple keys are used to search within a node and a small number of nodes are searched. Most database systems employ B+-Trees, a variant of the B-Tree.

In [2, 3], authors showed that B+-Trees have a better cache behavior than T-Trees. At the time T-Tree was proposed, cache behavior is not considered, since the gap between processor and main memory speeds was not that large. For most of the T-Tree nodes, only the two end keys are actually used for comparison. Additionally it has to store a record pointer for each key within a node, though most of the time the record pointers won’t be needed. This means that the utilization of each node is low. On the contrary, a B+-Tree uses multiple keys to search within a node. Also in each internal node B+-Trees store keys and child pointers, but the record pointers are stored on leaf
nodes only. Additionally while the branching factor of T-Trees is 2, that of B+-Trees is "number of keys within a node/2". Therefore if the node size of a B+-Tree and T-Tree is the same, a B+-Tree has the low height, cache misses, and high cache utilization, i.e. a B+-Tree is more cache conscious than a T-Tree.

[2, 3] suggested to fit a node size in a cache line, so that a cache load satisfy multiple comparisons. [2] introduced a cache sensitive search tree, which avoids storing pointers by employing the directory in an array. Although the proposed tree shows less cache miss ratio, it has a limitation of allowing only batch updates and rebuilding the entire tree once in a while. They then introduced an index structure called CSB+-Tree (Cache-Sensitive B+-Tree) that support incremental updates and retain the good cache behavior of their previous tree index structure [3]. Similar to their previous tree structure, a CSB+-Tree employs an array to store the child nodes. However, it now has one pointer for the first child node and the location of other child nodes is calculated by an offset to the pointer value. We used a similar approach to reduce the pointers within a node.

In addition, T. M. Chilimbi demonstrates placement mechanism, clustering [21] that improves cache performance by increasing a pointer structure’s spatial and temporal locality, and structure splitting [21, 22] that improves the cache behavior of structures larger than a cache block. We apply the clustering and splitting methods to T-trees when making T-trees cache-conscious.

3. CACHE SENSITIVE T-TREES

3.1 Cache Insensitiveness of T-Trees

The reasons that T-Trees are not quite effective to utilize the cache compared to other index structures such as B+-Trees are as follows. First, cache misses are rather frequent in T-Trees. The height of a T-Tree is much higher than the one of a B+-Tree. That is, the total number of memory accesses from the root to the leaf node is higher in T-Trees. Another reason that a T-Tree has higher cache misses is due that it does not align the node size with the cache line size. As shown in [2, 3, 20, 21], setting the node size with the cache line size is indeed desirable to decrease the cache miss of an index structure.

Secondly, in T-Trees, much portion of data brought to the cache is not actually used. Whenever the processor wishes to access a memory location for a node and the location is not found in a cache, i.e., cache miss, a copy of data for the location is transferred from memory into cache. This transfer incurs a delay since main memory is much slower than cache. Within the copied data in cache T-Trees use only two keys (maximum and minimum keys) for comparison and access another memory location for another node. In contrast, B+-Trees use \(|\log_2 n|\) keys that are brought to the cache for binary search comparison. Additionally, T-Trees use a record pointer for each key within a node, which leads the half of the node space is not utilized but wasted in the cache.

3.2 Cache Sensitive T-Trees

In this section we present the Cache Sensitive T-Trees and describe how we make the original T-Trees more cache-conscious by resolving cache-insensitiveness.
**Higher usage of cached data:** For T-Trees, the only data used for comparison within a node are its maximum and minimum keys. In a modified T-Tree [1], only maximum key is used for comparison. Therefore we can extract the maximum keys as hot fields applying structure splitting [21]. Then we construct a binary search tree which consists of only the maximum keys of each node (Fig. 2 (b)). We use the binary search tree as a directory structure to locate a node that contains an actual key that we are looking for. The size of the binary search tree is not big and great portion of it may be cached. More importantly, the cached data will be hit high since every searching explores the tree first.

![Diagram](image)

Fig. 2. (a) A T-tree; (b) The corresponding binary search tree with (a); (c) The corresponding CST-tree with (a).
Removal of pointers: First, if a binary tree is represented as an array, there is no need to store explicit pointers to the child or parent nodes. If a node is stored at index $i$ in an array and the root is at 1, then its parent, left and right child nodes may be found at $i/2$, $i \times 2$, and $i \times 2 + 1$, respectively. In [22], similar method of this is called clustering, which attempts to pack data structure elements likely to be accessed contemporaneously into a cache block. Secondly, when the child node groups of any given node group are stored contiguously as J. Rao [3] does, we need only one child pointer to indicate a first child node group explicitly (Fig. 2 (c)).

Alignment of node size with cache line size: We make a binary search tree as full as possible given an array which size is the same to the cache line. We call each binary search tree in an array a node group. For example, given that keys are 4 bytes integers, if a cache line size is 32 bytes, then a binary search tree in a node group may contain up to 7 keys and its height is 3 (Fig. 2 (c)). We always align the size of each node group with cache line size, so that there will be no cache miss when accessing data within a node group i.e., a child node to access is indexed $i \times 2$ or $i \times 2 + 1 < (\text{cache line size/pointer size})$. We use pointers to access from a node group to other node groups. Obviously, cache misses are unavoidable when accessing across the node groups.

Now we introduce our modified T-Tree, called Cache Sensitive T-Tree, as follows.

CST Trees: The CST-Tree is a $k$-way search tree which consists of node groups and data nodes (assume that a node group can have $k-1$ keys).

(P1) Data node contains keys and node group consists of maximal keys of each data nodes.
(P2) Each node group is a binary search tree represented in an array.
(P3) The tree is balanced, i.e., difference in height between the left and right sub-tree is less than 2, and a binary search tree of any node group is also balanced.
(P4) Sub-trees are also CST-Trees.

Fig. 3. Node structure of CST-trees.

Fig. 3 shows a node structure of CST-Trees, where we assume that a data node can contain $s$ keys and a node group can contain $n$ keys. A data node contains keys and the corresponding record IDs (RID), while a node group consists of maximal keys of data nodes and a pointer of a control block. The control block contains the corresponding pointers of data nodes, a pointer of parent, a height, and the child node groups. As com-
mented in a previous paragraph, in a CST-Tree the size of each node group is aligned with cache line size, so that there are no cache miss when accessing data within a node group. The child node groups of a node group are stored contiguously as well. Therefore \( i \)th child node group of a node group can be placed by accessing the address which is calculated as \( \text{control block address} + \text{number of data nodes} \times \text{size of data node pointer} + \text{size of parent pointer} + \text{size of height} + i \times \text{size of node group} \). When calculating the address of \( i \)th child node group, we implement the calculation using a constant to achieve performance since except \( i \) and \( \text{control lock address} \) the others can be determined at compile time.

### 3.3 Operations on a CST-Tree

In this section, we consider search, insert, delete and balancing algorithm on CST-Trees.

#### 3.3.1 Search operation

Search algorithm of CST-Trees is different from T-Trees, since CST-Trees consist of node groups and data nodes. An illustrative example of a CST-Tree is shown in Fig. 4.

![Fig. 4. An illustration of search operation in CST-trees.](image)

First, accessing the root node group incurs 1 cache miss. Since the given key ‘287’ is bigger than the key ‘160’, ‘240’, and ‘280’, we access the second node group and it incurs second cache miss. In the second node group, since ‘287’ is smaller than ‘300’, we mark the current comparing position. ‘287’ is also smaller than ‘290’, therefore we move the mark to the current comparing position. In the leaf node group, after the last key comparison, we do a binary search on the data node which corresponds to the last mark, and it incurs third cache miss. If there exists a given key at the data node, we have succeeded to find the search key. Otherwise we have failed.

```
CST_Search(key, tree)
// key: a key to find, tree: a CST-tree
// RID: a record ID to be found
compareKey = get the first key to compare in the root node group of tree;
// 1st step: traverse node groups
while (compareKey != NULL) {
    if (key <= compareKey) {
        lastMarkedNode = data node corresponding to current key;
        compareKey = get the key of left sub-tree;
    } else
        compareKey = get the key of right sub-tree;
}
// 2nd step: binary search in a data node
if (lastMarkedNode != NULL) {
    dataNode = get the data node from lastMarkedNode;
    RID = binary search in the dataNode;
    return RID;
} else return NOTFOUND;
End
```
During a search operation on CST-Trees from the root to the leaf node group, only accessing a sub-tree (child node group) and a data node incur cache misses. Doing a binary search in a node group does not incur a cache miss, because the size of a node group is the same as one of a cache line. Therefore the number of cache misses of a CST-Tree search operation is “CST-Tree height + 1” (3 cache misses in Fig. 4). We present the evaluation results for the number of cache misses on a search operation in section 4.2 and describe the time complexity in section 3.4.

3.3.2 Insertion/deletion operation

Insertion and deletion algorithms of CST-Trees are similar to T-Trees with an exception of a tree balancing algorithm (section 3.3.3).

An insertion operation is as follows. First, we find the data node to insert the given key and then insert the key to the corresponding data node. If the data node is not full, we simply insert the key to the data node. When the given key becomes a maximal key within the data node, we replace the key of the corresponding node group with the given key. If the data node is full, we delete the minimal key and insert the given key to the data node. Then we insert the deleted key into the left sub-tree as a maximal key. When there is no left sub-tree of the data node, we add a new data node (if we need to add a new node group, we have to add a new node group first) and insert the deleted key there.

As the balance check and rotation within the node group after the addition of a new data node is needed, balance check and rotation between node groups after the addition of a new node group is also needed. We present the rotation at section 3.3.3.

Fig. 5 is an example of insertion operation of inserting the key ‘288’ into a CST-Tree. In Fig. 5, the position to insert the new key is at the data node ‘A’ whose maximal key is ‘290’. Because the data node ‘A’ is full, we delete ‘286’ and insert the given key ‘288’.

![Diagram of insertion operation in CST-trees](image)
‘288’ into ‘A’. When we insert the deleted key ‘286’ to the data node ‘B’, we delete ‘281’ again and insert ‘286’ into ‘B’. Since there is no left sub-tree, we add a new node group (because there is no room in the leaf node group, we need to add a new node group), add new data node, and insert ‘281’ into the new data node.

We note here that deletion operation in CST-Trees is similar to the one in T-Trees except for a tree balance algorithm. Since it has analogy to opposite of insertion operation which is explained at the previous phrase, we do not describe the detail on deletion operation in CST-Trees.

3.3.3 Balancing algorithm

Balancing is a part of insertion/deletion operation to keep a CST-Tree balanced and is correspond to splitting/merging which is happened to in a B+-Tree. A CST-Tree is at whole a k-way search tree in which each node group contains binary search trees. For balancing the binary search trees we may apply a balancing algorithm similar to those of AVL Trees and T-Trees. A performance factor that we prioritize is the cache miss. Note that balancing a binary search tree does not cause a cache miss in that we align the node group size with the cache line size. However, every access to a non-cached node group causes a cache miss. Therefore we should pay more attention to balancing a CST-Tree across the node groups so as to minimize the average number of accesses to node groups.

Tree Balancing Across Node Groups

In this subsection, we explain how to balance a CST-Tree when a difference in height of its sub-trees goes beyond one. Albeit we present a detailed algorithm later, let us first see its basic operation, I-to-J rotation.

An I-to-J \((P, i, j)\) rotation is to move the \(i\)th child node group of \(P\) to the \(j\)th child node group. For example, Fig. 6 shows how a CST-Tree structure changes after an I-to-J rotation on \(P\) when \(i\) and \(j\) are 3 and 2, respectively. The other case that \(i\) is less than \(j\) is symmetrical to this case, so it is not shown.

In Fig. 6, \(Q\) is the 3rd child node group of \(P\), \(b\) is the 2nd child node group of \(P\). \(separator\) (‘2’) is the middle key of \(P\) between \(i\)th and \(j\)th child node group. We copy \(b\) to \(tempPrevJthChild\), move \(separator\) ‘2’ to \(Q\) (modified \(b\)) as the minimal key, and move all other keys of \(Q\) except for the maximal key to \(Q\). Then we move ‘5’ of the maximal
key of \( Q \) to the position of the previous separator, move \( \text{tempPrevJthChild} \) (b) to the 1st child node group of \( Q' \), and move the 1st (x), 2nd (y), 3rd (z) child node group of \( Q \) to \( Q' \) as 2nd, 3rd, 4th child respectively. Finally, we move \( w \) to \( P \) as the 3rd (ith) child node group.

A detailed node group balancing algorithm using the basic I-to-J rotation is illustrated in Fig. 7. In the algorithm, \( \min H(p) \) and \( \max H(p) \) mean the minimum and maximum value among the heights of the sub-trees that are children of a given node group \( P \). \( \min I(p) \) and \( \max I(p) \) are the array index values for the sub-trees that result in \( \min H(p) \) and \( \max H(p) \).

![Fig. 7. The CST-trees balancing algorithm for node groups.](image)

### 3.3.4 Time complexity

In this section, we discuss the time complexities of search, insertion, and deletion operations of CST-Trees. Let us say that \( n \) is the number of keys, \( s \) is the number of keys that a T-Tree contains within a node, and \( m \) is the number of keys that a node group of a CST-Tree contains or that a node of a B+-Tree contains.

If we store \( s \) keys into a data node, then the height of a \( m \)-way search tree to contain \( n \) keys should be at least \( \log_s(n/s) \). Each node group contains a binary search tree of which height is \( \log_m \). Search operation requires navigating a CST-Tree from the root node group to a leaf node group, and then again searches for a key within a data node. Then search operation requires \( \log_s(n/s) \times \log_2(m) \) to locate a target data node, \( \log_2 \) to find a key in a data node. Therefore, the time complexity of the search operation becomes \( O(\log n) \).

Our insertion operation of CST-Trees needs to locate a target data node to which a
key is inserted. If the target data node is already full, then the minimum key should be removed from the tree and inserted back into the left subtree of the target data node. In the worst case, an insertion operation requires $O(\log_2 n)$ to locate a target data node, $O(\log_2 s)$ to delete a key from the binary search tree, and $O(\log_2 n)$ to insert the key into the left subtree. Therefore, the time complexity of an insertion operation becomes $O(\log_2 n) + O(\log_2 s) + O(\log_2 n) = O(\log_2 n)$.

Our deletion operation also needs to locate a target data node where the key to be deleted is stored, and then it can delete the key from the target data node. Similar to the insertion operation, it needs additional operations to avoid the underflow of the tree. Therefore, the time complexity of a deletion operation becomes $O(\log_2 n)$.

Finally let us analyze the time overhead that a rotation requires for balancing a CST-Tree as the last step of an insertion or deletion operation. In CST-Trees, a binary search tree within a node group is an array structure. Therefore a rotation requires the memory copies of node groups that need to be relocated. A basic I-to-J rotation needs to move “number of child node groups + 2” number of node groups; i.e., a source node group ($I$) to be rotated, the child node groups of the source node group, and the target node group ($J$). For example, in Fig. 6, $Q$ is a source node group to be rotated. Then its child node groups ($x, y, z, w$) need to be moved by memory copies. In addition, $b$ (target node group) also needs to be moved. Assuming that a node group is 16 bytes and the array size is 3, we need to copy $6 \times 16 = 96$ bytes of data. Furthermore, if a cache line size is 64 bytes, then the array size of a node group becomes 15. And we need to copy $17 \times 64 = 1152$ bytes of data. Compared to this, in the same environment, a node split of CSB+-Trees needs to copy on average $8 \times 64 = 512$ bytes of data [3]. In short, the overhead for a CSB+-Tree node split is about the half of the one for a CST-Tree rotation.

4. PERFORMANCE EVALUATION

4.1 Experimental Environment

We performed an experimental comparison of the B+-Trees, T-Trees, CSB+-Trees, and the proposed CST-Tree. We implemented CST-Trees in C and the program was compiled and built by GNU cc compiler, which are available for every platform that we used in the experiment. For the implementation of CSB+-Trees and B+-Trees, we referred to the sources of [2, 3] that are proposed by original authors. Firstly, we implemented the source codes and tested on Sparc machines, then we modified some codes accordingly to the hardware platforms that were equipped with multi-core CPUs.

Fig. 8 is the list of the hardware platforms that we chose for experiment. Machine A and B are equipped with one dual-core CPU microprocessors of which architectures are different and manufactured by different corporations. The CPU processor contained in machine-A employs a shared L2 cache while one in machine-B employs separate L2 caches per core. Note that for comparative study we performed our experiment on hardware machines with single-core CPU as well. Both machine C and D are equipped with

---

1 We used a free-software to check the details of chipsets employed in machine A, B, and C. The program is available at http://www.cpuid.com/1, and the version we used is v1.39.


<table>
<thead>
<tr>
<th>Machine-A</th>
<th>Machine-B</th>
<th>Machine-C</th>
<th>Machine-D</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of CPU processors</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multi-Core? (No. of cores per processor)</td>
<td>Yes (2)</td>
<td>Yes (2)</td>
<td>No (1)</td>
</tr>
<tr>
<td>Cache structure</td>
<td>Shared L2 cache across dual cores</td>
<td>Separate L2 cache per core</td>
<td>Separate cache per processor</td>
</tr>
<tr>
<td>CPU clock speed</td>
<td>2.66GHz</td>
<td>2.0GHz</td>
<td>2.40GHz</td>
</tr>
<tr>
<td>L1 cache (&lt;cache size, cache line size&gt;)</td>
<td>2 × &lt;32K bytes, 64bytes&gt; (Data)</td>
<td>2 × &lt;64K bytes, 64bytes&gt; (Data)</td>
<td>&lt;8K bytes, 64bytes&gt; (Data)</td>
</tr>
<tr>
<td>L2 cache (&lt;cache size, cache line size&gt;)</td>
<td>&lt;4096K bytes, 64bytes&gt;</td>
<td>2 × &lt;512K bytes, 64bytes&gt;</td>
<td>&lt;512K bytes, 64bytes&gt;</td>
</tr>
<tr>
<td>RAM</td>
<td>2G bytes DDR2</td>
<td>1G bytes DDR2</td>
<td>1.5G bytes DDR</td>
</tr>
<tr>
<td>Operating system</td>
<td>Redhat Enterprise Linux ES v3</td>
<td>Redhat Enterprise Linux ES v3</td>
<td>Redhat Enterprise Linux ES v3</td>
</tr>
</tbody>
</table>

Fig. 8. The CPUs and their cache specifications of four different machines used in the experiment.

single-core CPU processors. Machine-C has one processor while machine-D has two processors.

In the original CSB+-Tree, node groups are allocated dynamically upon node split. Memory allocation calls can be saved if we pre-allocate the space for a full node group whenever a node group is created. CST-Trees also adopt a scheme to pre-allocate the whole space for a node group. In order to conduct a fair performance comparison, we also tested a variant of CSB+-Trees in which the whole space of a node group is pre-allocated when keys are inserted. In our insertion experiment, we call it CSB+- (full), while we call the original CSB+-Tree as CSB+- (org). For deletion, we used “lazy” policy as it is practically used [3].

We used the Valgrind debugging and profiling tool for Linux operating system and the Performance Analysis Tool for Sun operating systems, in order to measure the number of CPU cache misses. We only considered the L2 cache misses as they did in [3].

We set the keys and each pointer to be 4 bytes integers and 4 bytes in all experiments. All keys are randomly chosen within the range from 1 to 10 million. The keys are generated in advance to prevent the key generating time from affecting the measurements. The node sizes of all the methods are chosen to 64 bytes, same to the cache line size of each machine, since choosing the cache line size to be the node size was shown close to optimal [2, 3]. We repeated each test three times and report the average measurements.

---

2 Note that we do not include the actual model names of the microprocessors, since the purpose of our experiments is not to reveal the precise benchmark of each microprocessor.

3 We used the Valgrind 3.2.3 and the Sun ONE Studio 8. The Valgrind is freely available under GNU license at http://www.valgrind.org.
4.2 Results

Searching In the first experiment, we compared the search performance of the methods. We varied the number of keys and inserted all the keys into each index, and then measured the time and the number of cache miss that were taken by 200,000 searches. Each search key value was randomly chosen from the generated keys. Figs. 9-12 show the results. In figures, ‘# of keys’ means the number of keys contained in the tree.

![Graphs showing search performances in machine-A, machine-B, and machine-C.]  
(a) CPU elapsed time. (b) Cache misses.

Fig. 9. Search performances in machine-A (1CPU, dual-cores, and shared L2 cache).

Fig. 10. Search performances in machine-B (1CPU, dual-cores, and separate L2 caches).

Fig. 11. Search performances in machine-C (1CPU, single-core).

In general, CST-Trees show the best both in terms of speed and cache miss rate. CSB+-Trees, B+-Trees, and T-Trees follow the next in order. CST-Trees are on average (of performance improvements of 4 machines) 65.4%, 69.6%, and 77.7% faster than

4 We do not attempt to directly compare the performances of four microprocessors by drawing all graphs in a chart, since it may misguide some readers to directly consider the results as the performance benchmark of each microprocessor.

5 We use a relative performance ratio, i.e., (A – B)/A. For example, (elapsed time by CSB+ – elapsed time by CST)/elapsed time by CSB+. 
Fig. 12. Search performances in machine-D (2 CPU, single-core).

CSB+-Trees, B+-Trees, and T-Trees, respectively. CST-Trees also show the least number of cache miss among the methods. These performance differences come from gap of a tree height and the number of cache misses. As analyzed in section 3.3.4, time complexity of the CST-Tree search operation, $O(\log_2 n)$, is the same as CSB+-Tree and B+-Tree. But a tree height of CST-Tree that has a branching factor $m$ (assume $m$-way search tree and a data node may contain $s$ keys) is $\log_{m}(n/s)$. A CSB+-tree height is $\log_{m}(n/s)$, since a CSB+-Tree node has to store a child node pointer and keys, as well as the number of keys within the internal node. A B+-Tree height is $\log_{m}(n/s)$, since when storing keys, a B+-Tree stores not just a key but also corresponding child node pointer.

As a tree height gets higher, as a number of cache miss gets bigger.

We may observe two particular interesting results in these experiments. First, as the number of keys becomes larger in trees, the difference between CST-Trees and other methods in their cache miss numbers becomes larger too. Then among the methods, T-Tree shows steeper slope than others in its cache miss graphs, although the number of cache misses are linearly incremented as others. Secondly, the number of cache misses may greatly vary with the machine architectures. For example, in Fig. 9 (b), the average cache miss numbers of four trees on machine-A with 500K search keys is about 782K, while it is 2,278K and 2,276K on machine-B and C with same search keys, respectively. Note that the total L2 cache size of machine-A is 4 times bigger than B, and 8 times bigger than C, although their cache line sizes are the same to 64bytes. The machine-D that has a much larger L2 cache size significantly decreases the average number of cache misses for all cases. According to the result that both machine-B and C show a similar number of cache misses, just to have a double-cores without sharing the L2 cache may not affect the number of cache misses.

**Insertion and Deletion** In the next experiment, we tested the performance of insertion and deletion. Before testing, we first stabilized the index structure by bulk-loading 1 million keys, as they did in [3]. Then we performed up to 200K operations of insertion and deletion, and measure the elapsed time for the given number of operations (Figs. 13 (a) to 16 (b)).

In insertion tests, full CSB+-Trees show the best in insertion, while B+-Trees, CST+-Trees and T-Tree show comparable performance in their insertions. T-Trees are among the worst in machines except one (machine-D) where original CSB+-Trees also perform poor. As analyzed in section 3.3.4, since the overhead of rotation in a CST-Tree is larger than that of splits in a CSB+-Tree, CST-Tree insertion is slower than CSB+-Tree.
insertion. Furthermore, we may know that much time is consumed by inserting a given key and balancing a tree in insertion operation. Thus a B+-Tree and CSB+- (full) that
have less overhead for balancing is faster than a CST-Tree. On the point of split overhead, whenever it needs to split a node, since CSB+- (org) needs to reallocate a contiguous space for the whole sibling nodes and copy the sibling node data to new space, its insertion is slower than that of B+-Tree and CSB+- (full).

The deletion performance also showed a similar pattern to that of search. As mentioned earlier, we employed “lazy” strategy for deletion. Most of the time on a deletion is spent on pinpointing the correct entry in the leaf node. In deletion tests, CST-Trees show the best performance. CSB+-Trees, B+-Trees, and T-Trees follow the next in order.

5. DISCUSSIONS

5.1 CST-Tree Variation for Range Search

There is a T*-Tree [19] which is a variation of T-Tree. When doing range search, a T*-Tree has advantage of that the cost of traverse is less than a T-Tree. On range search, a T-Tree traverses unnecessary nodes because it has to traverse from the root node. However, as a T*-Tree traverses to the next node directly with the next pointer, it can reduce the traverse cost for processing a given range query. We may discuss about the variation of CST-Tree for range search as T*-Tree.

5.1.1 Index structure using a doubly linked list

We name CST+-Tree which is a variant of a CST-Tree to improve performance of range search using the previous and next pointers between data nodes. Each data node includes keys, record IDs of the keys, and the previous/next pointers. The linked list that links each data node with pointers enables range search to be efficient, although traditional CST-Tree is inefficient in range search.

The number of all data nodes in the given index tree is as follows. As a CST+-Tree uses a pointer pair which is the previous pointer and the next pointer instead of a key-record pair, there are \( s - 1 \) key-record pairs in a data node. Therefore, CST+-Trees require \( n/(s - 1) \) data nodes to store \( n \) key-record pairs, while CST-Trees require \( n/s \) data nodes. When \( s - 1 \) keys are stored in a data node, the height of a CST+-Tree is \( \log_m(n/(s - 1)) \) because the structure of a node group of a CST+-Tree is the same as a CST-Tree.

5.1.2 Range search

In this section, we discuss the range search operation on a CST+-Tree. Note that the other operations of a CST+-Tree is similar as a CST-Tree. The range search algorithm may consist of two steps which are as follow.

Step 1: Find a position of a data node which contains the minimum key of the given keys as a query.

Step 2: Do range search with a linked list from the data node found in step 1.
In step 1, to find a position of a data node containing the minimum key, the minimum key of a query is compared with the keys of node groups. If the given key is greater than the keys of a node group, the algorithm traverses the right child node group. If the given key is less than one of the keys, we can assume that the corresponding data node contains the given key or the given key exists in the left child node group of the node. When traversing the left child node group, we mark the current node of the node group before traversing. If the given key is not found in data nodes of a leaf node group, we find the given key in the data node of the latest marked position. In the second step, the algorithm searches query keys from the data node that contains the minimum key of the given keys as a query, to the next data nodes which are followed by the next link. If the compared key is equal to or greater than the maximum key of the given query keys, the range search is completed.

6. CONCLUSION

In this paper, we proposed a new index structure called CST-Tree. CST-Trees are obtained by applying cache consciousness to T-Trees. Our time complexity analysis and experimental results show that CST-Trees provide much better performance than other existing main memory index structures owing to the better cache behavior.

Our experimental study shows that cache sensitive trees provide much better performance than their original versions. In searching operations, CST-Trees perform better than CSB+, B+-Trees, and T-Trees. CSB+-Trees also show good performance than B+-Trees. CST-Trees also show comparable performance on insertion operations and better performance on deletion operations, although the performance benefits are less than in searching. As the gap between CPU and memory speed becomes widening, CST-Trees should be considered as a replacement of T-Trees in future.

The experiment is worthy because the experimental results show that cache sensitive index structures may benefit of the designs of modern commodity processors. It is, however, limited in that we have not developed an analytical model of our cache sensitive index on a multi-level shared cache architecture, so that we can mathematically compare the empirical results to the theoretically-expected behavior of the model. This should be one of the works we shall deal with in future.

It is one of the hottest research topics in database community to tune a database management system to perform well enough to benefit the commodity microprocessors. We are trying to not only make index structures more cache-conscious, but also apply parallelism to index structure operations for efficient utilization of multi-core microprocessors [23].

REFERENCES

2. J. Rao, et al., “Cache conscious indexing for decision-support in main memory,” in

Ig-Hoon Lee received the B.S. and M.S. degrees in Computer Science from University of Seoul, and the Ph.D. degree from Seoul National University, Seoul, Korea, in 2005. He is currently a research Professor at School of Computer Science and Engineering in Seoul National University, Seoul, Korea. From 2001 to 2008, he was a director of the Department of Research and Development of Prompt Inc., Seoul, Korea. His research interests include database systems, mobile systems, product information management, ontology, and e-business technology.

Sang-Goo Lee received the B.S. degree in Computer Science from Seoul National University. He also received the M.S. and Ph.D. in Computer Science from Northwestern University at Evanston, Illinois, USA, in 1987 and 1990. He is currently a Professor in School of Computer Science and Engineering at Seoul National University, Seoul, Korea. He has been the director of Center for e-Business Technology at Seoul National University. His current research interests include e-business technology, mobile applications, database systems, and ontology.

Junho Shim received his M.S. degree from Seoul National University, and the Ph.D. degree in Computer Science from Northwestern University at Evanston, Illinois, U.S.A., in 1998. He is currently a Professor at Department of Computer Science in Sookmyung Women’s University, Seoul, Korea. Previously he held assistant professor position with Drexel University, Pennsylvania, U.S.A. His research interests include database systems, data warehousing, product information management, and ontology.