Vol. 47 No. 1 IPSJ Journal Jan. 2006

Regular Paper

Array-based Cache Conscious Trees

Hidehisa Takamizawa,[†,☆] Kazuyuki Nakajima[†,☆☆]
and Masayoshi Aritsugi[†]

Making effective use of cache can give good performance. In this paper, Array-Based Cache conscious trees (ABC trees for short) are proposed for realizing good performance of not only search operation but also update operation. The logical structure and manipulation of an ABC tree are similar to those of a B^+-tree. The initial space of an array for an ABC tree as it is supposed to be a complete tree is allocated. This allows the tree to have contiguous memory space for its core and to reduce the number of pointers in it. As a result, the key capacity of a node increases and we can make effective use of cache. We also present an enhancement of ABC trees, which can increase the capacity of an ABC tree with overflow nodes. We describe how we can decide whether to create an overflow node when a node overflows for performance. Some experimental studies show that ABC trees can give good performance of operations under certain conditions.

1. Introduction

It has become possible for us to have computers with large main memories recently. A large amount of data tend to be memory resident when processing the data on such a computer. The processes on data can be performed very quickly if the data have been already loaded onto main memory from secondary storage. Researches on main memory databases are expected to be applied in various fields.

In such an environment, memory accesses would become a bottleneck like disk accesses are conventionally a bottleneck in database applications. The gap between CPU speed and main memory access speed is now large, and it is said that the gap will grow larger with time. If we can exploit cache effectively we will realize better performance.

One of the effective ways of using cache is to reduce cache misses. While the capacity of cache is generally small compared with main memory, cache access speeds are much faster than memory access speeds. The cache is used for reducing the gap in accessibility between register in CPU and main memory by holding a part of the contents of main memory. If data to be processed by CPU has existed in cache, processes can be performed efficiently. However, if the data has not been in cache, the cache miss occurs and CPU must access the data on main memory, which takes much more time than accessing only the cache. We can thus improve the efficiency if we can reduce the number of cache misses.

There have been studies on effectively using caches. Shatdal, Kant and Naughton enhanced algorithms of query processing in relational databases in order to improve the performance by making use of the cache [12]. Ailamaki, et al. realized a cache-conscious data organization for storing relations in relational databases [1]. They split a page into several pieces, and data for an attribute are stored in each of the pieces. There have also been proposals of new cache conscious indexing techniques discussed as follows.

Conventional indexing techniques are mainly for reducing disk I/O's. Recently, new indexing techniques have been investigated for cache conscious manipulation of data, e.g., cache-sensitive search trees (CSS-trees) [10], cache sensitive B^+-trees (CSB^+-trees) [11], cache-conscious R-trees (CR-trees) [7], partial-key trees [2], and prefetching B^+-trees (pB^+-trees) [4]. One of the key ideas to realize cache conscious indexing techniques is to reduce cache misses. To do this, CSS-trees [10] and CSB^+-trees [11] reduce the number of pointers that a node holds, thereby increasing the data capacity of a node and the amount of memory resident data. CR-trees [7] and partial-key trees [2] realize this by compressing keys. pB^+-trees [4] make use of prefetching instruction provided by modern microprocessors so that they can hide the performance impact of cache misses.

† Gunma University
☆ Presently with Toshiba Solutions Corporation
☆☆ Presently with ACCESS CO., LTD.

In this paper, we propose another new indexing structure called Array-Based Cache conscious trees, or ABC trees [9),13)]. The logical structure and manipulation of an ABC tree are similar to those of a B^+-tree [5)]. The core structure of the tree is implemented with an array, which is a contiguous space in memory, and it allows us to reduce the number of pointers in it. When navigating through the core of an ABC tree, the position of a child node can be obtained by calculating spatial relations among nodes. Due to the reduction of the number of pointers and the memory allocation, ABC trees achieve not only efficient search operation but also efficient delete and insert operations under certain conditions. We also present an enhancement of ABC trees with overflow nodes. This achieves a large increase of data capacity but also efficient processing of operations. We investigate the way of deciding whether to create an overflow node when a node overflows for performance. Moreover, we discuss conditions where ABC trees can give good performance and present experimental results for showing the efficiency of the trees.

The remainder of this paper is organized as follows. Section 2 mentions related work and compares with our work. Section 3 introduces core ABC trees and an enhancement of them, and describes their structures and procedures of operations. Section 4 shows experimental results and discusses the performance of ABC trees. Section 5 concludes this paper.

2. Related Work

There have been many studies on management of data taking account of effective usage of cache for improving efficiency (e.g., Refs. 1)∼4), 6), 7), 10)∼12) and 14)). In those, most similar studies to ours are CSS-trees [10)] and CSB^+-trees [11)] because they improved the performance by proposing new cache conscious indexing structures for reducing the number of pointers in a node.

Cache sensitive search trees (CSS-trees) [10)] improve the performance of search processing in OLAP environments. The tree structure is implemented by using an array, and no pointer exists in its tree structure. A node can therefore hold more data than a node of any other tree structures. A node can be identified with location information in the array. It proposed the way of aligning the size of a node to cache line size. This enables us to access a node with at most one cache miss. CSS-trees need less space and achieve much more efficient search operations than B^+-trees [5)] and T-trees [8)]. In addition, the whole of a CSS-tree corresponds to an array that is allocated a contiguous space in memory. As a result, CSS-trees achieve high efficiency of search operation. Although CSS-trees can work well in such environments as OLAP, it is very hard to implement efficient updates on them.

Cache sensitive B^+-trees (CSB^+-trees) [11)] are proposed as a cache conscious version of B^+-trees. The main goal of CSB^+-trees is to realize efficient update operations, since CSS-trees cannot provide them. While an internal node of B^+-trees holds pointers to all its child nodes, only a pointer to the first child node is held in an internal node of CSB^+-trees. Child nodes of an internal node are managed as a node group. A node group is allocated contiguously in memory space, and a node in a node group can thus be addressed with the pointer to the group and the offset from the head of the group to the node. The size of a node in CSB^+-trees is aligned to cache line size like in CSS-trees. Note that the number of data in a node in CSB^+-trees is less than that in CSS-trees because a node in CSB^+-trees holds a pointer and an integer expressing the number of data that the node holds. Note also that a CSB^+-tree is not allocated in a contiguous space generally, because every node group in a CSB^+-tree is created completely dynamically. Rao and Ross examined several variations of CSB^+-trees, and concluded that full CSB^+-trees, in which necessary space for storing a full node group is allocated beforehand, can give the best performance of operations including search, insert, and delete, even though they have essential space overheads [11)]. In other words, they concluded that it is practical for a good cache conscious tree to have a margin for space when space overhead is not a big concern.

In this paper, we propose ABC trees that have good features of both CSS-trees and CSB^+-trees. Similar to CSS-trees, a core ABC tree is constructed with an array, which is a contiguous space in memory; a node in a core ABC tree does not hold any pointer and is identified with location information in the array. Moreover, ABC trees can give good performance of update operations, like CSB^+-trees, by means of having a margin for space. We also propose an enhancement of the structure of a core ABC

tree for handling overflow nodes, thereby ABC trees can grow dynamically. Note, however, that some conditions have to hold for allowing ABC trees to give good performance.

3. ABC Trees

In this section, we introduce the structure and manipulations of ABC trees. First, we discuss core ABC trees, which are the core structure and manipulations of ABC trees. Then, we show an enhancement of ABC trees, which can have overflow nodes. Finally, we discuss conditions where ABC trees can give good performance.

3.1 Core ABC Trees

A core ABC tree is implemented with an array and no pointer is included in its internal structure. Rao and Ross showed that preallocation of space for a node group to have a capacity of full-data nodes increased the performance of update processes [11]. According to the results, the space for a core ABC tree is preallocated as it is complete, and moreover the whole tree is allocated contiguously in memory.

In this paper we suppose that when bulkloading a core ABC tree keys are sorted and the total number of keys is known. Also, there is no multiple occurrences of a key.

3.1.1 The Structure

A core ABC tree is constructed of an array, which has the capacity of nodes in a complete tree. A node, which corresponds to a part of the array, stores keys and an integer expressing the number of keys located in the node. Every node is assigned its node ID, which starts from 0 for the root node in order of breadth first, and is located in the array in this order. **Figure 1** shows node ID's of a core ABC tree, an internal node of which has two keys and three child nodes☆, and the height is three. The upper side of the figure shows the logical structure of the tree and the lower side depicts the physical structure. Note that, similar to CSS-trees [10], core ABC trees do not have any pointers in their structures. An internal node of the core ABC tree does not have any pointers, while an internal node of the CSB$^+$-tree [11] has a pointer pointing to the head of its child nodes.

Figure 2 shows a core ABC tree, an internal node of which has two keys, with 18 keys in total. Since the number is the same as the key capacity of the tree, all nodes are filled in with keys. Every internal node has three $(= 2 + 1)$ child nodes.

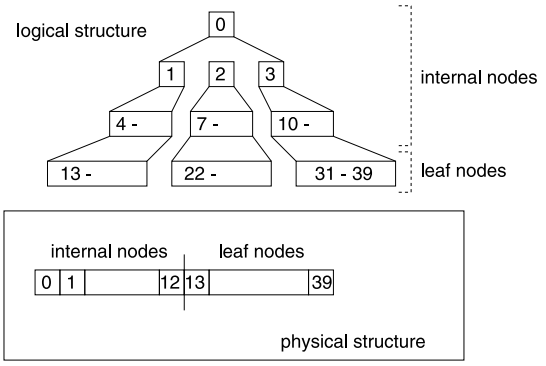

Fig. 1 Node ID's of a core ABC tree.

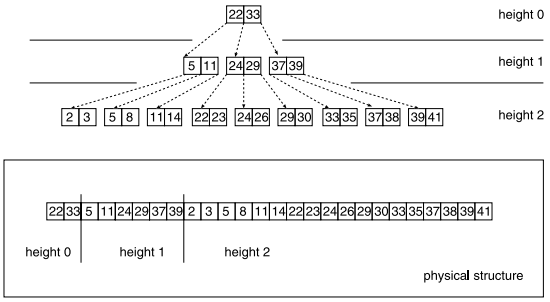

Fig. 2 A core ABC tree with 18 keys.

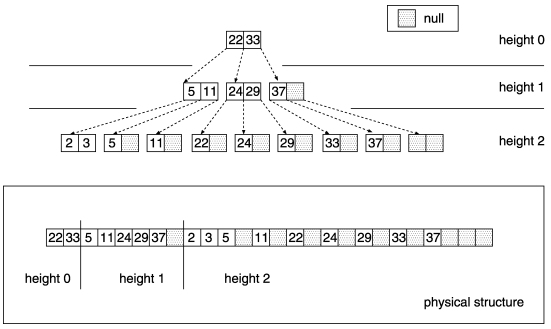

Fig. 3 A core ABC tree with 9 keys.

Figure 3 shows another core ABC tree, an internal node of which has two keys as well, with 9 keys in total. In general the number of keys that we have to manage is not equal to the key capacity of an ABC tree. Thus, trees like the figure exist in the real world. Note that the last leaf node shown in the figure is necessary for the tree, even though it does not have any keys, because the tree must be complete. We shall discuss locations of null keys and null nodes later.

☆ For simplicity, we omit in figures in this paper that every node has an integer expressing the number of keys stored in the node.

3.1.2 Node ID's

When bulkloading an ABC tree, we allocate space for it as if it is a complete tree even though some key fields are not filled in with keys. All nodes are thus stored contiguously in an array. Maintaining the whole spatial relations among nodes enables us to get the location of a node without following pointers. This leads us to eliminate pointer completely from an internal node of a core ABC tree. Let m be the number of keys that an internal node can hold. The capacity of children that an internal node can have in this case is $m+1$. The ID range of a child node, of which ID is $childID$, of node, of which ID is $nodeID$, is the following.

$$\begin{aligned} & nodeID \times (m+1)+1 \\ & \quad \le childID \\ & \quad \le (nodeID+1) \times (m+1) \end{aligned} \tag{1}$$

Thus, ID of the i-th child node ($0 \le i \le m$) is expressed as follows.

$$nodeID \times (m+1) + i + 1 \tag{2}$$

Given a node ID $nodeID$, ID of the parent node is expressed as follows.

$$\lceil nodeID/(m+1) \rceil - 1 \tag{3}$$

It should be noted that, since a core ABC tree is complete, we can navigate the tree by using the node ID's, even though there is no pointer in its structure.

3.1.3 Operations on a Core ABC Tree

Here we consider bulkload, search, insert, and delete operations on a core ABC tree. In the following, let n and m be the total number of keys to be stored in a core ABC tree and the number of keys that an internal node can hold, respectively.

3.1.3.1 Bulkload

When bulkloading an ABC tree, which is actually a core ABC tree, we put keys in leaf nodes and also put keys in internal nodes simultaneously, if necessary, that is, we do not divide bulkloading into filling in leaf nodes and internal nodes with keys. Since a core ABC tree is constructed as if it is complete, given a key, the internal node that should hold the key can be found uniquely by the location of it at the leaf level. In other words, we can find the internal node for storing the key without navigating any internal nodes, resulting in at most one cache miss for doing that.

Let $height$ be the height of the ABC tree to be bulkloaded. The following condition, where LN stands for the minimum number of leaf nodes necessary for storing n keys, should hold.

$$(m+1)^{height-1} < LN \le (m+1)^{height} \tag{4}$$

Since a node has the capacity of m keys, $LN = \lceil n/m \rceil$. Thus, we can obtain $height$ as follows.

$$\begin{aligned} height &= \lceil \log_{m+1} LN \rceil \\ &= \left\lceil \log_{m+1} \left\lceil \frac{n}{m} \right\rceil \right\rceil \end{aligned} \tag{5}$$

We can distinguish each node by its node ID, which begins with 0 at root node in breadth first order, as shown in Fig. 1. The total number of nodes of the ABC tree, TN, is expressed as follows.

$$\begin{aligned} TN &= \sum_{i=0}^{height} (m+1)^i \\ &= 1 + (m+1) + (m+1)^2 + \\ &\quad \cdots + (m+1)^{height} \\ &= \frac{(m+1)^{height+1} - 1}{m} \end{aligned} \tag{6}$$

For bulkloading we put keys in ascending order to leaf nodes. The ID's of leaf nodes are between $\{(m+1)^{height} - 1\}/m$ and $\{(m+1)^{height+1} - 1\}/m - 1$.

We also put keys to internal nodes appropriately while putting keys to leaves. In Fig. 2, for example, keys 5 and 11 are put in both leaf nodes and their parent node, while key 2 does not appear in the bottom level of internal nodes. In fact, given a key, at the time when we put it at the leaf level we can decide whether or not it should also be put in an internal node in a core ABC tree, by using the algorithm shown in **Fig. 4**. For example in Fig. 2, key 11 is the fourth key location at the leaf level. Since $4 = 2 \times (2+1)^0 \times 2$ and $2\ div\ (2+1) = 0$ and $2\ mod\ (2+1) = 2$, key 11 is copied to the 0-th internal node at the next higher level from the

(Let w be an integer of greater than or equal to 0, and x, y, and z be natural numbers.)
if the location of the key is the $m \times (m+1)^w \times x$ leaf
  and $y = x\ div\ (m+1)$ and $z = x\ mod\ (m+1)$
  then
    the key is copied to
    the y-th internal node
    at the $w+1$ higher level from the leaf node
    as the z-th key
endif

Fig. 4 Algorithm for putting a key into the index part.

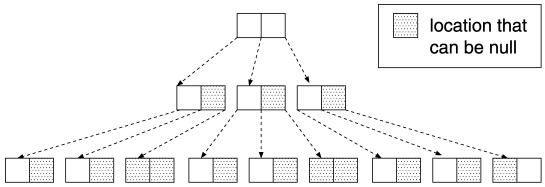

Fig. 5 Locations that can be null in ABC tree (m=2).

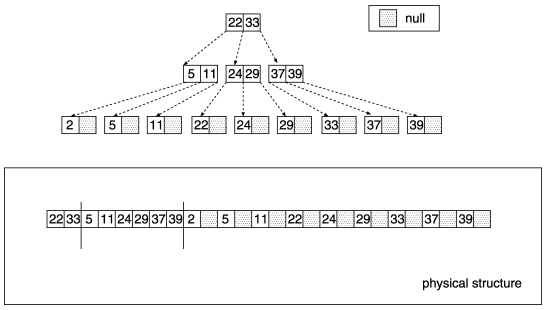

Fig. 6 ABC tree (m=2) with 9 keys after bulkloading.

leaf node as the 2nd key.

In order to make a core ABC tree complete, we may put null keys and/or null nodes, which have no keys, in the tree. We should take care of where to put null nodes in the tree, because it will affect the efficiency of manipulating the tree. In the index part of a core ABC tree we do not want to create any null nodes or to put null keys at any level except the bottom level. We also do not want to put null keys at the first position of nodes at the bottom level of the index part, for allowing to access leaf nodes easily. We therefore do not put keys at the leaf level in such locations where $w = 0$ and $z = 0$ in Fig. 4. **Figure 5** shows locations that can be null in a core ABC tree where $m = 2$ and the height is 2.

Figure 6 shows the result core ABC tree by bulkloading 9 keys. As shown in the figure, we scatter locations of null when bulkloading as much as possible in this paper. This is the reason why Fig. 3 and Fig. 6 are different. There are several alternatives about how we locate null keys, and locations of null keys can affect the performance of manipulations of ABC trees. We, however, do not discuss this problem in this paper and it is included in our future work.

3.1.3.2 Search

Logical procedure of search operations on core ABC trees is similar to that on B^+-trees. It begins with accessing root node, of which node ID is 0. We determine which child node be accessed next by binary searching the node. Let $nodeID$ be the ID of the node currently accessed. If the i-th child node is the next node to be accessed, then the child node is navigated by using its ID, i.e., $nodeID \times (m + 1) + 1 + i$. This navigation is performed repeatedly until reaching a leaf node. If a leaf node is visited, the search key is searched in the node. This is carried out by binary search, too. If the key is found then the search finishes as successful, otherwise does as unsuccessful. Not to mention, the integer expressing the number of keys of a node is used in binary search on the node.

3.1.3.3 Insertion

If the number of keys is less than $(m + 1)^{height} \times m$ in a core ABC tree, there are one or more empty slots for a key to insert in the tree, even if the key cannot be stored a leaf node because it is full. On the other hand, if a tree stores $(m+1)^{height} \times m$ keys, no more key can be inserted unless the tree can obtain more space. That is, we can insert a key without increasing the height of the tree in the former case, while we need to increase it for inserting a key in the latter case.

When a core ABC tree has less than $(m + 1)^{height} \times m$ keys, we first search the leaf node in which the key to insert should be. If the leaf node has room, put the key into the node appropriately. If the target node has no room, then we find room in another leaf node close to the target node, shift keys from the target node in the direction to the neighbor node in order to make room in the target node, and insert the key. Note that when shifting keys to make room in the target node we may also have to update keys in the index part of the tree accordingly to the shift. This is done in the same way as in bulkloading shown in Fig. 4.

When the number of keys we have to manipulate becomes more than or equal to $(m + 1)^{height} \times m$, we cannot insert any key into the tree by the way described above. This is because the structure of a core ABC cannot grow incrementally. In addition, we need to consider not only the number but also the other factors for making ABC trees more efficient. We will discuss this issue in the following subsections.

3.1.3.4 Deletion

Since a core ABC tree makes use of null keys, the manipulation of underflows is different from that in a B^+-tree. In a general B^+-tree at least $\lceil (m+1)/2 \rceil$ of pointers in an internal node must be used and at least $\lceil (m + 1)/2 \rceil$ keys must be filled in a leaf node. On the other hand,

there may be more null keys and null nodes in a core ABC tree in order to keep it complete. Note that if too many null keys appear in a core ABC tree, it is hard to maintain it properly and is likely to make the performance worse. We therefore decided that the number of keys appearing in a core ABC tree must not be less than nor equal to $(m+1)^{height}+1$, and if the condition does not hold we reduce the height of the tree by one.

If a key is deleted from a core ABC tree and this deletion does not make the number of keys in the tree less than $(m+1)^{height}+1$, then we do not reduce the height of the tree. We first search the key to be deleted in a leaf node. Then, we delete the key from the leaf node. If the key also appears at the parent node, then it should be deleted and the node should be maintained properly. If the deletion does not make the leaf node null, the delete operation finishes. If the leaf node does not have any keys, it should be checked whether the deletion affects the index part of the tree or not. That is, if the key to be deleted appears in a lowest internal node and is the last key of the node, then the deletion does not affect the index part of the tree and thus the delete operation finishes. If the deletion affects the index part, we find a neighbor leaf node which has more than one key or which has one key and can become a null leaf node without affecting the index part (Fig. 5), and shift keys from the neighbor leaf node to the leaf node which had the key to be deleted. We may need to update the index part according to the shift.

If a key is deleted from a core ABC tree and this deletion reduces the number of the keys in the tree to less than $(m+1)^{height}+1$, then the deletion makes it hard to maintain the index part properly. In this case we reduce the height of the tree by one by means of shifting keys in leaf nodes, except for the key to be deleted, to the lowest internal nodes. This shift is performed by scanning leaf nodes and bulkloading keys. Internal nodes of the lowest level of the original tree become leaf nodes of the new tree. The height of the result tree is therefore one lower than that of the original tree. The space, which was for leaf nodes, is not deleted but allocated for the case where we need to increase the height of the tree. We, however, did not implement this in the system used for evaluation, because it is obvious that the performance of this is bad. It is included in our future work.

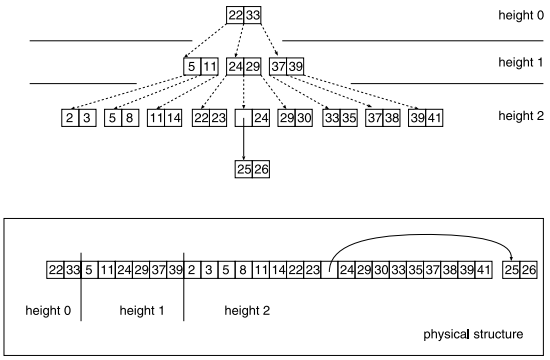

Fig. 7 An ABC tree with an overflow node.

3.2 An Enhancement of ABC Trees

As described in the previous subsection, a core ABC tree is implemented with an array. Since the space for a core ABC tree is allocated in a contiguous space and the tree does not have any pointer for maintaining its structure, we expect that the tree can give good performance.

However, there are several problems on core ABC trees. Since a core ABC tree is implemented with an array, which is fixed size, we need to rebuild it to an array with one level higher for keeping the structure when more data than those expected at the time of initializing the tree have to be manipulated. It must take naturally long time to do this. Another problem is that when there is skew of locations of empty cells in a core ABC tree and a large number of insertions to areas where almost all cells are filled occur the performance of the core ABC tree must degrade.

Here we propose an enhancement of core ABC trees for overcoming such problems. The key idea is to allow trees to have overflow nodes. **Figure 7** shows an ABC tree with an overflow node, which is the result of inserting key 25 to the tree shown in Fig. 2.

As shown in Fig. 7, a key field in a leaf node having an overflow node is used for holding a pointer to the overflow node. If an overflow node needs another overflow node, then the field for the number of keys is used for holding a pointer to the next overflow node. In a chain with a leaf node and overflow node(s), keys are filled in from the leaf node. For checking whether the value in the field stands for the number of keys or the address, the most significant bit is used.

3.2.1 Operations on the ABC Trees

All operations except insert operation on the ABC trees are almost the same as those on the

core ABC trees described before. Bulkload operations on the two trees are identical, that is, the initial form of the enhanced ABC tree is a core ABC tree. Searching a key on the ABC tree begins with the same procedure described before for finding the leaf node in which the key should be. If the search key is equal or less than the maximum key in the leaf node, then the node is searched. Otherwise, its overflow node(s) is accessed and searched. Delete operation on the ABC tree has an additional procedure concerning overflow nodes; we delete an overflow node and the pointer to it when the overflow node is going to have no key after the deletion.

Since the ABC trees are allowed to have overflow nodes, we have to decide whether we should create an overflow node at the time when inserting a key to a filled leaf node. This is processed by comparing the costs of insert operations by shifting and by creating an overflow node.

When considering the cost for insert operations to ABC trees with overflow nodes, we have to take account of not only one insert operation that is going to be processed but also search and insert operations following the insert operation. For example, if one insert operation creates an overflow node for a leaf node, then a following search operation on the leaf node might need an additional node fetch, a following insert operation on the leaf node could be processed only by shifting a small number of keys, or a following insert operation on a leaf node near to the leaf node could be processed with less key shifts than the case before creating the overflow node.

We therefore analyze the costs for one insert operation and following $(m-2)$ insert operations in this study, where m is the number of keys that a leaf node can hold. We suppose that the ratio of the numbers of search operations to insert operations to be performed is s to 1. We thus study the costs for one insert operation and the following $(m-2)$ insert and $s \times (m-1)$ search operations. Delete operations are not taken into account because the cost for delete operations can nearly be the same as the cost for search operations.

In the following, we describe the results of the cost analysis; the detailed analysis can be found in Appendix.

Let *height* be the height of the tree. We call the leaf node that is to be processed by the first insertion *pivot* in this paper. Let d be the number of nodes that we access from *pivot* for finding room in one direction. Note that overflow nodes, which are described later, are counted in d, if any. Let $fetchcost$, $shiftcost$, and $newcost$ be the times for fetching a node, shifting a key, and allocating memory for a node, respectively. Note that it is easy to measure these values of a given environment by running a simple program. Let $usage = \frac{\#\ keys\ stored}{key\ capacity} \times 100$ and $w = \left\lceil \frac{m-1}{m \times \frac{100-usage}{100}} \times \frac{1}{2} \right\rceil$. Then, the cost with no overflow node is expressed as

$$\begin{aligned} & fetch_s \times fetchcost + key_s \times shiftcost \\ & \quad + (m-1) \times (d + w - 1) \times \frac{m}{m+1} \\ & \quad \times shiftcost \\ & \quad + s \times (m-1) \times height \times fetchcost \end{aligned} \tag{7}$$

where $fetch_s$ and key_s are the expected number of nodes to be fetched and the expected number of keys to be shifted for the $(m-1)$ insertions, respectively.

On the other hand, the cost with an overflow node is expressed as

$$\begin{aligned} & fetch_o \times fetchcost \\ & \quad + key_o \times shiftcost + newcost \\ & \quad + s \times (m-1) \times height \times fetchcost \\ & \quad + s \times (m-1) \times \frac{1}{2 \times d + 1} \\ & \quad \times \frac{1}{m} \sum_{k=1}^{m} \frac{k}{m+k-1} \times fetchcost \end{aligned} \tag{8}$$

where $fetch_o$ and key_o are the expected number of nodes to be fetched and the expected number of keys to be shifted for $(m-1)$ insertions, respectively.

We can obtain the best value of d with Eqs. (7) and (8). When we insert a key to a full leaf node of the ABC tree, we try to find room in leaf nodes that are in the area of d from the leaf node. If there is room in the area, we shift keys and make room in the target leaf node and insert the key to the node. If we cannot find any room in the area, then we choose a leaf node, which is in the area and has the least number of overflow nodes among the leaf nodes in the area, create an overflow node and attach it to the leaf node, shift keys to the overflow node with inserting the key appropriately.

3.3 Discussion

As presented before, the idea of ABC trees is simple but works well due to its good characteristics with regard to cache usage. However,

there are conditions under which ABC trees can really work well. Here we mention them in order to make it easy to exploit ABC trees.

The number of keys that the current implementation of an ABC tree can manage would be roughly restricted by constant times capacity of a corresponding core ABC tree. Although an ABC tree can basically manipulate a large number of keys by means of the enhancement discussed in Section 3.2, there are cases where ABC trees cannot give good performance, e.g., a case where a long chain of overflow nodes is created. To avoid such cases, we would need to find time for maintaining ABC trees to reduce the number of overflow nodes of them and for reconstructing them to make them core ABC trees. Applying ABC trees to some real applications and implementing how to maintain them according to the characteristics of each application will be included in our future work.

As discussed in Section 3.2, we need values of $fetchcost$, $shiftcost$, $newcost$, and s for deciding whether we should create an overflow node when processing split of a node. It is easy to measure the values of $fetchcost$, $shiftcost$, and $newcost$ by running a simple program on a machine on which ABC trees will use in advance. On the other hand, we need to estimate the value of s appropriately. We think it would be possible by holding statistics concerning operations.

If there is skew of key values to be inserted, the performance of ABC trees would be worse than other trees including CSB$^+$-trees, because the skew will cause a large number of key shift processes and/or a large number of overflow node creations in an ABC tree resulting in a long chain of overflow nodes. We think it would be solved by using an appropriate hash function that can avoid such skew.

4. Experimental Evaluation

4.1 An Experimental Environment

To show the efficiency of ABC trees, we constructed four trees, namely, B$^+$-trees, CSB$^+$-trees, core ABC trees, and ABC trees described in Section 3.2, and evaluated them on a workstation, the configuration of which is shown in **Table 1**.

We implemented ABC tree, CSB$^+$-tree, and B$^+$-tree in C++. In the implementation, operator "`new`" was used for allocating space of a core structure of ABC tree, of a node of CSB$^+$-tree and B$^+$-tree, and of an overflow node of ABC tree.

Table 1 Testbed configuration.

machine type	Sun Blade 1000
cpu	UltraSPARC-III (750 MHz)
memory	
size	2048 MB
block size	64 B
interleaving	4-way
OS	Solaris 8
cache	
L1 size	64 KB
L1 line size	32 B
L1 associativity	4-way set
L2 size	8192 KB
L2 line size	256 B
L2 associativity	direct mapped
compiler	Forte Developer 7 C++ 5.4 2002/03/09
compiler option	-xtarget=ultra3 -xcache=64/32/4:8192/256/1 -xarch=v9 -xO5

According to the conclusion of Ref. 11) that full CSB$^+$-trees is the best choice when space overhead is not a big concern, we implemented full CSB$^+$-trees in the experiments. Codes for the experiments were all written in the C++ programming language. The experiments run with a 64-bit kernel on the machine. The sizes of a key and a pointer were both eight bytes. **Table 2** shows key capacities of internal nodes and leaf nodes in the core ABC tree, B$^+$-tree, and CSB$^+$-tree, where the size of a node was aligned to L2 cache line size (Table 1). In the core ABC tree, a node consists of keys and an integer expressing the number of keys that the node holds. Dissimilar to the other trees, the structures of an internal node and leaf node of the core ABC tree are the same. In the CSB$^+$-tree, an internal node consists of keys, a pointer to the first child node, and an integer expressing the number of keys that the node holds, and a leaf node consists of keys, two pointers to neighbor nodes, and an integer expressing the number of keys that the node holds ☆. In the B$^+$-tree, an internal node consists of keys, pointers to child nodes, and an integer expressing the number of keys that the node holds, and a leaf node consists of keys, two pointers to neighbor nodes, and an integer expressing the number of keys that the node holds. Note

☆ Implementation details may be slightly different from those written in Ref. 11), especially concerning structures of nodes. However, we believe that the basic idea is identical and that the performance results of CSB$^+$-trees show the performance characteristics of the original work in Ref. 11).

Table 2 Key capacities of a node.

core ABC		CSB$^+$		B$^+$	
internal node	leaf node	internal node	leaf node	internal node	leaf node
31	31	30	29	15	29

Table 3 Characteristics of initial trees.

ABC		CSB$^+$		jammed-CSB$^+$		B$^+$	
height	usage (%)	height	usage (%)	height	usage (%)	height	usage (%)
4	61.5	5	56.4	4	100.0	6	69.8

$$usage = \frac{\#\ keys\ stored}{key\ capacity} \times 100$$

that a node of the ABC tree had the most key capacity (Table 2), since there was no physical pointer in the core ABC tree, although the capacity of a node decreased when the node had to hold a pointer to an overflow node.

4.2 Experiments and Results

We created trees with 20,000,000 keys for initialization. For the initial construction, we prepared sorted keys of the number. The keys were natural numbers made at random. A core ABC tree was bulkloaded with the sorted keys in the algorithm shown in Section 3.1.3.1, and was used as the initial tree of both a core ABC tree and an ABC tree in the experiments. A B$^+$-tree was initially prepared by constructing a B$^+$-tree with two keys randomly selected out of the keys and then inserting the rest keys to the B$^+$-tree. We prepared two types of full CSB$^+$-trees, expressed as CSB$^+$-trees and jammed-CSB$^+$-trees in this paper. A CSB$^+$-tree was prepared by bulkloading with the sorted 2,000,000 keys randomly selected from the 20,000,000 and then inserting the rest 18,000,000 keys randomly. This was done in the same way in Ref. 11), and created a balanced CSB$^+$-tree. A jammed-CSB$^+$-tree, the other type of CSB$^+$-trees in this paper, was prepared by bulkloading the 20,000,000 sorted keys. This method created a CSB$^+$-tree in which all keys were arranged densely and there were few vacancies in leaf nodes. **Table 3** shows characteristics of these initial trees, where height stands for the height of each tree. We can see that leaf nodes of the jammed-CSB$^+$-tree were almost all filled in; in fact, all except for the right most node of jammed-CSB$^+$-tree were filled in. As shown in Table 3, the ABC tree and the jammed-CSB$^+$-tree had the lowest heights.

Figure 8 shows the times required for search operations. The results show that the core ABC tree could give the best performance and the ABC tree was the next. The reason why the performance of ABC tree was slightly worse

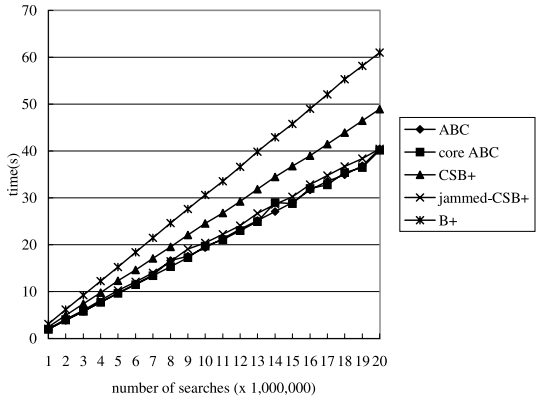

Fig. 8 Performance comparison of search operations.

than that of the core ABC tree is that the procedure for handling overflow nodes is necessary for ABC tree even though there was no overflow node at this moment. Since the height of the jammed-CSB$^+$-tree was lower than that of the CSB$^+$-tree, it is natural that the jammed-CSB$^+$-tree outperformed the CSB$^+$-tree. The reason why the ABC tree outperformed the jammed-CSB$^+$-tree might be that the cost of binary search in a node of ABC tree was smaller than that of the jammed-CSB$^+$-tree while the heights of both trees were the same in the experiments.

Figure 9 shows the times required for delete operations. For delete operations, we did not consider underflow of data in a node of CSB$^+$-tree and B$^+$-tree as in Refs. 4) and 11) in the experiments. When deleting a key from a node having more than one key we just deleted the key. When deleting the last key from a node we also deleted the node. The results show that the ABC trees outperformed the other trees except for the case of 19,000,000 deletions. The reason why the cost for 19,000,000 delete operations on the ABC tree become so large is that it needed a large number of key shifts for the operations. The times for 19,000,000 deletions from the ABC and core ABC trees were about

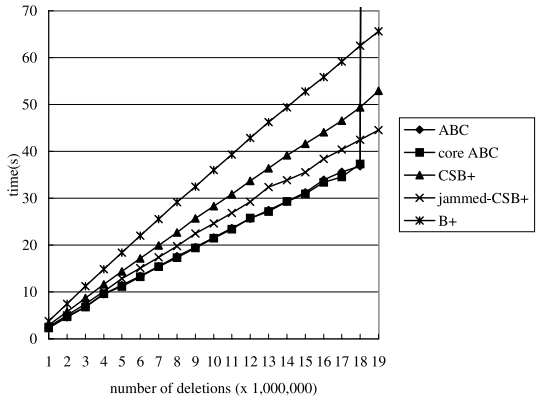

Fig. 9 Performance comparison of delete operations.

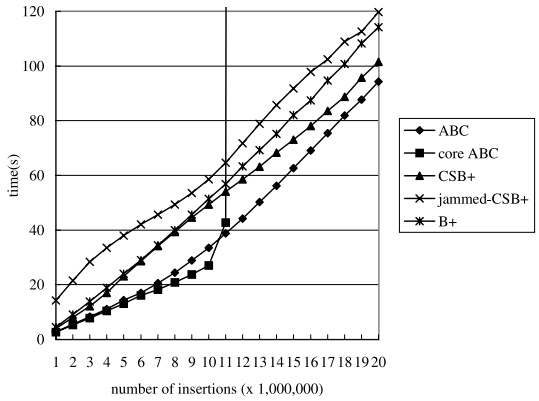

Fig. 10 Performance comparison of insert operations.

1395 and 1351 seconds, respectively. This problem will be included in our future work.

Figure 10 shows the times required for insert operations. We assumed that the value of s appearing in Eqs. (7) and (8) was 1 in the experiments. Due to the capacity of the core ABC tree in the experiments, we could not do insertions over 13,000,000 to the tree. In the experiments of insertions, the core ABC tree could give the best performance from 1,000,000 to 10,000,000 insertions and the ABC tree could give the best performance for the other cases. The time for 12,000,000 insertions to the core ABC tree was about 11445 seconds. On the other hand, since the ABC tree was allowed to have overflow nodes, the performance of the tree was good for all cases of insertions. The jammed-CSB$^+$-tree gave the worst performance of insert operations. This is because a lot of split occurred when inserting keys into the jammed-CSB$^+$-tree.

Figure 11 shows the times required for search operations after the insertions. This experiment was done for showing that our strategy described in Section 3.2 can work for combinations of insert and search operations. The keys to be searched were those used for the insertions, thereby highlighting the cost for manipulating overflow nodes in the ABC tree. In this experiment, the core ABC tree could give the best performance from 1,000,000 to 12,000,000 searches and the ABC tree could give the best performance in the rest cases. Dissimilar to the results shown in Fig. 8, the performance of the CSB$^+$-tree was almost the same as that of the jammed-CSB$^+$-tree. This is because the usage of a node of the jammed-CSB$^+$-tree became small by the insertions. Note that the ABC tree, which held some overflow nodes created by the insertions, outperformed the jammed-CSB$^+$-tree and the CSB$^+$-tree in this experiment. Moreover, we can notice that the calculation of d described in Section 3.2 worked well by combining the results shown in Fig. 10 and Fig. 11.

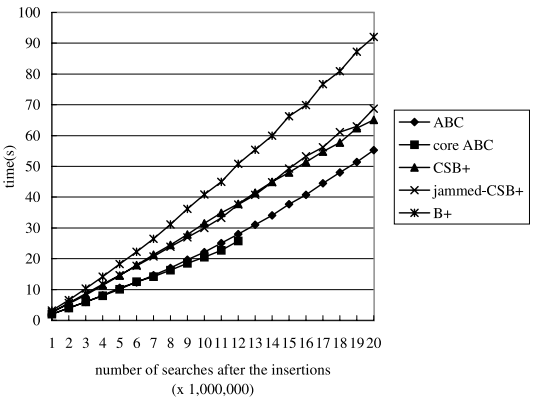

Fig. 11 Performance comparison of search operations after the insertions.

In order to demonstrate further the effectiveness of the calculation of d described in Section 3.2, we did another experiment where insert operations and search operations after the insertions were examined with varying the value d from $d-5$ to $d+5$.

Figure 12 shows the times required for insert operations with varying the value of d. The results show that the costs for the cases of $+2$, $+3$, $+4$, and $+5$ were larger than those for the rest cases. The performance of the rest cases was almost the same. This would be because the obtained value of d was 0 or nearly 0 in the experiments.

Figure 13 shows the times required for search operations after the insertions with varying the value of d. In this experiment, the

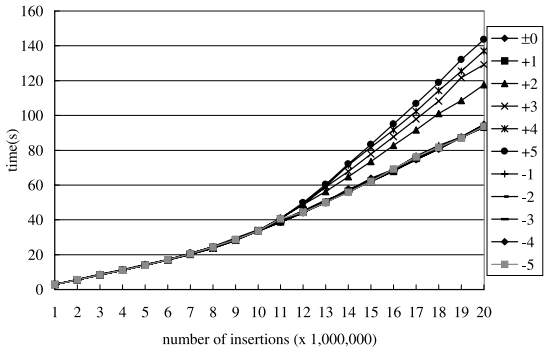

Fig. 12 Performance comparison of insert operations with varying d.

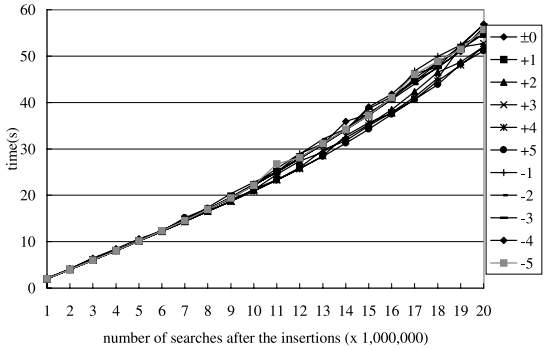

Fig. 13 Performance comparison of search operations after the insertions with varying d.

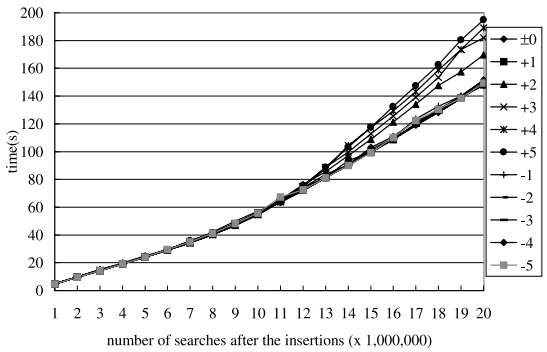

Fig. 14 The sums of the results shown in Fig. 12 and Fig. 13.

costs for the cases of +2, +3, +4, and +5 were smaller than those for the rest cases, that is, the results were opposite to those shown in Fig. 12. This is because when the value of d was large it tended not to create overflow nodes and thus the cost for search operations on such trees became lower.

Figure 14 shows the sums of the results shown in Figs. 12 and 13. With the results shown in Figs. 12, 13 and 14, we can conclude

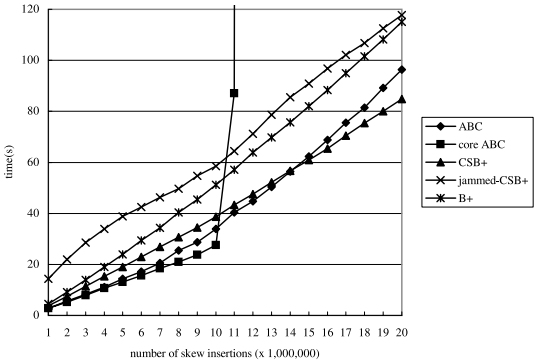

Fig. 15 Performance comparison of skew insert operations.

that the calculation of d described in Section 3.2 worked well.

The results we have shown so far were obtained in an environment where key values to be treated were distributed uniformly. Next, we show other results of experiments running in an environment where there was skew of key values to be inserted. We set that 50% of insert operations were to the area of central 1/3 range of the initial tree. **Figure 15** shows the results of the experiment. In this case, the ABC tree could outperformed the CSB$^+$-tree when the number of insertions was small. On the other hand, when the number was larger than 14,000,000 keys, the CSB$^+$-tree outperformed the ABC tree; the larger the number of insert keys was, the larger the gap between them became. This would be because the number of overflow nodes of the ABC tree became larger as the number of insert keys became larger. If the skew area was narrower, then the gap appeared with smaller number of insert keys and grew rapidly. We can also see that the degradation of performance of the core ABC tree in the case of Fig. 15 occurred when inserting smaller number of keys than the case of Fig. 10.

Figure 16 shows the times required for search operations after the skew insertions. The performances of the CSB$^+$-tree shown in Fig. 11 and Fig. 16 were almost the same, while the performance of the ABC tree shown in Fig. 16 was worse than that in Fig. 11. Thus we have to say that ABC trees would be influenced by skew of key values more easily than CSB$^+$-trees.

5. Conclusion

We proposed Array-Based Cache conscious trees (ABC trees) in this paper. Because a core ABC tree is implemented with an array,

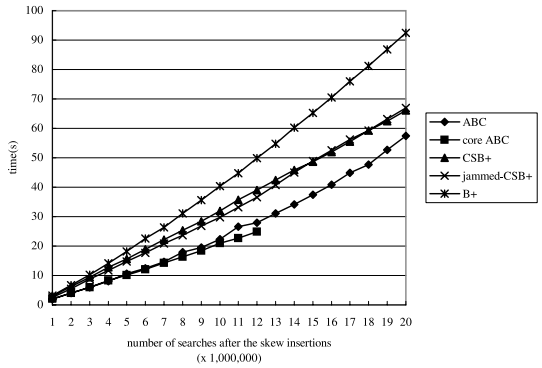

Fig. 16 Performance comparison of search operations after the skew insertions.

the space of the tree is allocated in contiguous memory space. By means of aligning the size of a node in an ABC tree to the cache line size, we can make effective use of caches with the ABC tree. We also studied an enhancement of core ABC trees for allowing them to have overflow nodes. Experimental results showed that ABC trees can give good performance under certain conditions by comparing with conventional trees.

In future, we will examine handling overflow nodes for more efficiency. Particularly, we need some mechanism for preventing an ABC tree from having a long chain of overflow nodes. Also, concurrency control mechanisms for ABC trees are included in our future work. In addition, applying ABC trees to some real applications, examining their availability, and investigating how to maintain and reconstruct ABC trees efficiently will be included in our future work.

Acknowledgments We would like to thank Yujiro Ichikawa for his help with the experiments. We would also like to thank the editor and anonymous reviewers for their valuable comments and suggestions on this paper.

References

1) Ailamaki, A., DeWitt, D.J., Hill, M.D. and Skounakis, M.: Weaving Relations for Cache Performance, *Proc. 27th International Conference on Very Large Data Bases*, pp.169–180 (2001).
2) Bohannon, P., McIlroy, P. and Rastogi, R.: Main-Memory Index Structures with Fixed-Size Partial Keys, *Proc. ACM SIGMOD International Conference on Management of Data*, pp.163–174 (2001).
3) Cha, S.K., Hwang, S., Kim, K. and Kwon, K.: Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems, *Proc. 27th International Conference on Very Large Data Bases*, pp.181–190 (2001).
4) Chen, S., Gibbons, P.B. and Mowry, T.C.: Improving Index Performance through Prefetching, *Proc. ACM SIGMOD International Conference on Management of Data*, pp.235–246 (2001).
5) Comer, D.: The Ubiquitous B-Tree, *ACM Computing Surveys*, Vol.11, No.2, pp.121–137 (1979).
6) Hankins, R.A. and Patel, J.M.: Data Morphing: An Adaptive, Cache-Conscious Storage Technique, *Proc. 29th International Conference on Very Large Data Bases*, pp.417–428 (2003).
7) Kim, K., Cha, S.K. and Kwon, K.: Optimizing Multidimensional Index Trees for Main Memory Access, *Proc. ACM SIGMOD International Conference on Management of Data*, pp.139–150 (2001).
8) Lehman, T.J. and Carey, M.J.: A Study of Index Structures for Main Memory Database Management Systems, *Proc. 12th International Conference on Very Large Data Bases*, pp.294–303 (1986).
9) Nakajima, K. and Aritsugi, M.: An Implementation of Array-Based Cache Conscious Trees with Overflow Nodes (in Japanese), *Proc. DBWeb 2003*, pp.125–132 (2003).
10) Rao, J. and Ross, K.A.: Cache Conscious Indexing for Decision-Support in Main Memory, *Proc. 25th International Conference on Very Large Data Bases*, pp.78–89 (1999).
11) Rao, J. and Ross, K.A.: Making B^+-Trees Cache Conscious in Main Memory, *Proc. ACM SIGMOD International Conference on Management of Data*, pp.475–486 (2000).
12) Shatdal, A., Kant, C. and Naughton, J.F.: Cache Conscious Algorithms for Relational Query Processing, *Proc. 20th International Conference on Very Large Data Bases*, pp.510–521 (1994).
13) Takamizawa, H. and Aritsugi, M.: Cache Conscious Trees using Arrays: A Proposal (in Japanese), *DBSJ Letters*, Vol.1, No.1, pp.11–14 (2002).
14) Zhou, J. and Ross, K.A.: Buffering Accesses to Memory-Resident Index Structures, *Proc. 29th International Conference on Very Large Data Bases*, pp.405–416 (2003).

Appendix

In this appendix, we describe the detailed analysis for obtaining Eqs. (7) and (8).

We suppose that the ratio of the numbers

of search operations to insert operations to be performed is s to 1. Let m be the number of keys that a leaf node can hold. We call the leaf node that is to be processed by the first insertion *pivot* in this paper. Let d be the number of nodes that we access from *pivot* for finding room in one direction. Note that overflow nodes, which are described later, are counted in d, if any. Let $fetchcost$, $shiftcost$, and $newcost$ be the times for fetching a node, shifting a key, and allocating memory for a node, respectively.

Let us first consider the cost without creating an overflow node for the operations. In this case, there is no empty slot in the leaf nodes within the distance d from *pivot*. Thus we need to shift keys to the leaf node(s) that is further than d from *pivot*. Let $usage = \frac{\#\ keys\ stored}{key\ capacity} \times 100$ and the expected number of nodes in which keys have to be shifted for the $(m-1)$ insertions be $d+w$, then w can be expressed as follows.

$$w = \left\lceil \frac{m-1}{m \times \frac{100-usage}{100}} \times \frac{1}{2} \right\rceil \tag{9}$$

When calculating d we suppose $usage$ as a fixed value during the operations, since the number of data stored in a tree is much larger than m.

Let $height$ be the height of the tree. The expected number of nodes to be fetched, $fetch_s$, and the expected number of keys to be shifted, key_s, for the $(m-1)$ insertions can be expressed as follows.

$$\begin{aligned} fetch_s = {} & (m-1) \times height + 2 \times d + 1 \\ & + (m-2)\Bigg(\\ & \frac{1}{2 \times d + 1} \left(2 \times (d + w - 1) + 1\right) \\ & + \frac{2}{2 \times d + 1} \\ & \times \sum_{k=w}^{d+w-1} \left(2 \times (k-1) + \frac{3}{2}\right)\Bigg) \end{aligned} \tag{10}$$

$$\begin{aligned} key_s = {} & \frac{m}{2} + m \times \frac{100 - usage}{100} \times \frac{1}{2} \\ & + m \times d + (m-2)\Bigg(\\ & m \times \frac{100 - usage}{100} \times \frac{1}{2} + \frac{m}{2} \\ & + \frac{1}{2 \times d + 1} \left(m \times (d + w - 1)\right) \\ & + \frac{2}{2 \times d + 1} \sum_{k=w}^{d+w-1} m \times (k-1)\Bigg) \end{aligned} \tag{11}$$

We need updates on internal nodes caused by shift operations except for the shift between the first child leaf node of an internal node to the last child leaf node of its left neighbor internal node. The probability of the occurrence of this update at a leaf node is $\frac{m}{m+1}$. The number of leaf nodes that we are considering is $d+w-1$. Thus, the number of updates on internal nodes is

$$(m-1) \times (d + w - 1) \times \frac{m}{m+1}\,. \tag{12}$$

Since an update in the index part copies a data stored in a leaf node, the cost for it can be estimated as $shiftcost$. Note that no other cost is necessary because it is expected that an internal node to be updated has already been cached by traversing the path from the root node to the leaf node.

The number of nodes to be accessed for a search operation is equal to the height of the tree. Then, the cost with no overflow node is expressed as follows.

$$\begin{aligned} & fetch_s \times fetchcost + key_s \times shiftcost \\ & \quad + (m-1) \times (d + w - 1) \times \frac{m}{m+1} \\ & \quad \times shiftcost \\ & \quad + s \times (m-1) \times height \times fetchcost \end{aligned} \tag{13}$$

This is Eq. (7).

Next, we consider the cost with an overflow node. As described before, let d be the number of nodes that we access from *pivot* for finding room in one direction. If we add an overflow node to *pivot* for one insertion, the overflow node has $(m-2)$ empty slots at this moment. Thus, $(m-2)$ insertion operations following the insertion operation are performed by shifting data to *pivot* or a leaf node which is d far from *pivot*. Let $d+y$ be the expected number of nodes that are between *pivot* and the node to which data are shifted by an insert operation. Let us define x $(x < d)$ as follows.

$$x = \left\lceil \frac{d+y}{2} \right\rceil \tag{14}$$

Then, we can calculate y as follows.

$$y = \left\lceil \frac{(m-2) \times \frac{d-x+1}{2 \times d+1}}{m \times \frac{100-usage}{100}} \times \frac{1}{2} \right\rceil \tag{15}$$

We can obtain x and y with Eqs. (14) and

(15).

Then, the expected number of nodes to be fetches, $fetch_o$, and the expected number of keys to be shifted, key_o, for $(m-1)$ insertions can be expressed as follows.

$$\begin{aligned} fetch_o = {} & (m-1) \times height + 2 \times d \\ & + (m-2)\Bigg(\\ & \frac{2}{2 \times d + 1} \sum_{k=1}^{x-1} \left(2 \times (k-1) + \frac{3}{2} \right) \\ & + \frac{2}{2 \times d + 1} \\ & \times \sum_{k=y}^{d+y-x-1} \left(2 \times (k-1) + \frac{3}{2} \right) \Bigg) \end{aligned} \tag{16}$$

$$\begin{aligned} key_o = {} & m \\ & + (m-2)\Bigg(\\ & \frac{2 \times x + 1}{2 \times d + 1} \times \frac{m + 1 + \frac{2 \times m - 1}{2}}{2} \\ & + \frac{2 \times d - 2 \times x}{2 \times d + 1} \\ & \times m \times \frac{100 - usage}{100} \times \frac{1}{2} \\ & + \frac{2 \times d}{2 \times d + 1} \times \frac{m}{2} \\ & + \frac{2}{2 \times d + 1} \sum_{k=1}^{x-2} m \times k \\ & + \frac{2}{2 \times d + 1} \sum_{k=y-1}^{d+y-x-2} m \times k \Bigg) \end{aligned} \tag{17}$$

Because we can express the possibility of access *pivot* in the search operations as $s \times (m-1) \times \frac{1}{2 \times d + 1}$, the cost with overflow nodes is expressed as follows.

$$\begin{aligned} & fetch_o \times fetchcost \\ & + key_o \times shiftcost + newcost \\ & + s \times (m-1) \times height \times fetchcost \\ & + s \times (m-1) \times \frac{1}{2 \times d + 1} \\ & \times \frac{1}{m} \sum_{k=1}^{m} \frac{k}{m + k - 1} \times fetchcost \end{aligned} \tag{18}$$

This is Eq. (8).

(Received April 4, 2005)
(Accepted October 11, 2005)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.2, pp.25–38.)

Hidehisa Takamizawa received his B.E. and M.E. degrees in Computer Science from Gunma University, Japan, in 2001 and 2003, respectively. He is presently with Toshiba Solutions Corporation, Japan. His research interests include database systems and security. He is a member of IPSJ.

Kazuyuki Nakajima received his B.E. and M.E. degrees in Computer Science from Gunma University, Japan, in 2002 and 2004, respectively. He is presently with ACCESS Co., Ltd., Japan.

Masayoshi Aritsugi received his B.E. and D.E. degrees in computer science and communication engineering from Kyushu University, Japan, in 1991 and 1996, respectively. Since 1996, he has been working at the Faculty of Engineering, Gunma University, Japan, where he is now an Associate Professor. His research interests include database systems and parallel and distributed data processing. He is a member of IPSJ, IEICE, IEEE-CS, ACM, and DBSJ.