Implementation of B-Trees Using Dynamic Address Computation

By H. Rex Hartson and Raymond T. West, Jr.

TR 90-37

IMPLEMENTATION OF B-TREES USING DYNAMIC ADDRESS COMPUTATION

Raymond T. West, Jr. and H. Rex Hartson*
Virginia Polytechnic Institute and State University

The B-tree is probably the most popular method in use today for indexes and inverted files in database management systems. The traditional implementation of a B-tree uses many pointers (more than one per key), which can directly affect the performance of the B-tree. A general method of file organization and access (called Dynamic Address Computation) has been described by Cook that can be used to implement B-trees using no pointers. A minimal amount of storage (in addition to the keys) is required. This paper gives a detailed description of Direct Address Computation and the resulting B-Tree implementation. The performance of the resulting system is analyzed, leading to the conclusion that, while the approach results in a simple implementation of B-trees, more work is required to achieve competitive performance for large B-trees.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.2 [**Database Management**]: Physical design—*Access methods*; H.2.4 [**Database Management**]: Systems; E.5 [**Files**]: Organization/structure

General Terms: algorithms, performance

Additional Key Words and Phrases: B-trees, ragged arrays, storage structures, dynamic addressing, pointer-free addressing, secondary indexing, file organization

---

* Authors' addresses: Raymond T. West, Jr., Computer Science Department, John Brown University, Siloam Springs, Arkansas 72761. H. Rex Hartson, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061.

1. INTRODUCTION

The B-tree is an m-way search tree which has been proposed and used for indexes and inverted files in database management systems [1, 3, 5]. The traditional implementation of a B-tree uses many pointers (more than one per key), which can directly affect the performance of the B-tree. This happens because the space required for the pointers reduces the number of keys that can be stored in a node. This reduces m, the order of the tree, and thus increases the number of levels in the tree. Since the number of disk accesses required to search a B-tree is the same as the number of levels, the performance cost is obvious. Cook has proposed a general method of file organization and address computation [4] and has suggested that it can be used to implement B-trees using no pointers. A minimal amount of storage (in addition to the keys) is required by this method, which he calls Dynamic Address Computation. Cook's dissertation contains a detailed discussion of the principles of Dynamic Address Computation and briefly introduces the concept of "pointer-free" B-trees with an algorithm to search a B-tree. This paper describes a complete implementation of the Dynamic Address Computation algorithms. Using the DAC procedures, it describes an implementation of a "pointer-free" B-tree management package, including searching of the tree and insertion and deletion of keys. Analytical performance measures are derived in an attempt to understand the performance characteristics of a B-tree in an implementation using Dynamic Address Computation.

Definition: An order m B-tree, T, is an m-way search tree that is either empty or has the following properties [6]:

1) The root node has a least two children.

2) All nodes other than the root node and the leaf nodes have at least $\lceil m/2 \rceil$ children.

3) All leaf nodes are at the same level.

There are a number of variations on this theme (B+-trees, B*-trees, Prefix B-trees, special handling of root nodes, key compression, etc. See [3]). The algorithms here build a basic B-tree. The extension to B-tree variations is obvious, and is not precluded in the implementation.

The data associated with a key value can be stored in the node with the key, but, in order to keep the degree of the tree high (and thus the depth low), the data is usually stored separately. This can be accomplished by storing a pointer to the data in the node with the key, since the pointer is usually smaller than the data itself. Alternatively, nothing at all might be stored with the key. In this case, when the key is found, the traversal of the tree continues until a leaf node is reached. This node, instead of a null pointer, would contain a pointer to the data associated with the key. If the entire tree is traversed without the key being found, the position in the leaf node defines the correct place to insert the key, if desired. The data is now stored in data nodes, which can be considered special nodes which are not subtrees (This results in what Horowitz and Sahni [6] call a B'-tree, and Comer [3] calls a B+-tree, but, for simplicity, the generic term B-tree will continue to be used).

When all of the data nodes in a B-tree of order m are at level x+1, N, the number of key values (and data nodes) can be shown to be within the following bounds [6]:

$$2[m/2]^{x-1} - 1 \leq N \leq m^{x} - 1$$

Conversely, for a B-tree of order m with N data nodes and key values, the maximum level of a non-data node is:

$$x \leq \log_{[m/2]} \{(N + 1)/2\} + 1$$

A search of the B-tree requires only x disk accesses (plus the access to the data). Since larger values of m produce exponentially smaller values of x, maximizing m is very important in the implementation of B-trees. At the same time, because of physical storage limits, nodes cannot grow arbitrarily large. Also, the transmission time of a block from disk to primary memory increases with the block size, and it is desirable to keep that small. In summary, the more keys that can be stored in a given amount of space, the better the performance of the B-tree.

2. ANOTHER MODEL FOR B-TREES

The previous definition of a B-tree strongly suggests an implementation, i.e., nodes containing keys and associated pointers to the child nodes. However, pointers have several disadvantages. The relative amount of space they occupy can be significant, especially if the keys are short. They force the designer either to reduce the degree of the B-tree or to increase the node size. Both are undesirable. In this section, a different model of a B-tree will be presented. It will suggest a different implementation, one which is compatible with Dynamic Address Computation.

2.1 The Ragged Array

The underlying structure for this model is a 2-dimensional "ragged array." This is an array in which the number of columns in a row can vary from row to row. For example, in Figure 1 array A is a ragged array with four rows and from one to six columns in each row.

				column			
	1	2	3	4	5	6	7
Row							
1	A(1,1)	A(1,2)					
2	A(2,1)						
3	A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)	A(3,6)	
4	A(4,1)	A(4,2)	A(4,3)				

Figure 1: A Ragged Array

Of course, this can be stored in a regular rectangular array where the column dimension is the largest required by any row, but that will waste a large amount of space when the rows are not full.

A better implementation of a ragged array would eliminate the wasted space; for example, consider this linear storage of A:

A(1,1),A(1,2),A(2,1),A(3,1),A(3,2),A(3,3),A(3,4),..,A(4,3).

This requires the storage of auxiliary information (row and column) and there is an increased cost for accessing the data. To find an existing data element requires some sort of search, and inserting a new element (e.g., A(2,2)) requires the movement of data.

A linked organization commonly used for sparse arrays can also be used. This uses a set of row pointers to point to the first element of each row, with links to subsequent elements in the row [7]. Again, pointers must be stored and links followed to find a given element.

This section will show a way that ragged arrays can be used to represent B-trees, but will not discuss their implementation, as that is the subject of later sections.

2.2 The B-tree as a Ragged Array

A B-tree can be represented with a ragged array by storing only the keys of each node as a row in the array (no pointers). Each node of the B-tree is denoted by an integer value which represents its row position in the array. That integer is called the node index. The term node i will be used to denote the node whose node index is i.

The keys within a node are denoted by the column index in the array, called the key index. Thus, for a ragged array named B_tree, B_tree(i,j) is the j-th key in the i-th node of the tree.

The nodes of the B-tree are not ordered in one of the common traversal orders (pre-order, post-order or in-order). Instead, they are ordered and numbered as follows:

1) The root node is node one, i.e. the first row in the ragged array.

2) All of the nodes in the second level follow in order from left to right. That is, in left to right order as defined by the keys in the root which are associated with these nodes.

3) The level three nodes follow in order, again from left to right as defined by the order of the keys in the level two nodes.

4) And so on, for all levels of the tree.

Also, the data which is being indexed by the B-tree is assumed to be stored in a separate array, DATA(data_index). An element of DATA may be a complex structure, divided into fields, a list of pointers, etc. The important point is that a datum is identified by an integer index which will be determined by the B-tree algorithms (this index is not physically stored anywhere in the system described here, but rather, is computed dynamically).

Some other information will also be needed:

N_keys(i):
the number of keys in node i.

N_nodes:
the number of nodes in the B-tree.

N_tree_levels:
the number of levels in the B-tree.

m:
the maximum number of subtrees of any node (the order of the B-tree).

Putting a B-tree into a ragged array in this way gives the configuration shown in Figure 2, where each line is one node.

```
B_tree(1,1)          ... B_tree(1,N_keys(1))                 Root

B_tree(2,1)          ...B_tree(2,N_keys(2))                    -
                      .                                        |
                      .                                   Level 2
                      .                                        |
B_tree(N2+1,1)       ... B_tree(N2+1,N_keys(N2+1)              -

B_tree(N2+2,1)       ... B_tree(N2+2,N_keys(N2+2))             -
                      .                                        |
                      .                                   Level 3
                      .                                        |
B_tree(N3+N2+1,1)    ... B_tree(N3+N2+1,N_keys(N3+N2+1))       -
                      .
                      .
                      .
                     etc.
```

Where N2 = N_keys(1)+1 is the number of level 2 nodes.

N3 = N_keys(2)+1 + N_keys(3)+1 ... + N_keys(N2+1)+1

is the number of level 3 nodes.

etc.

Figure 2: Abstract B-tree to Ragged Array Mapping

The way that this structure can be used to represent a B-tree is suggested by the observation that the number of nodes at level two is one greater than the number of keys in the root. The number of nodes at level three is the sum of the number of keys at level two plus the number of level two nodes. In general, the number of nodes at a level (i) is recursively defined as:

1) There is one node at level 1.
2) For 2 <= i <= N_tree_levels:

$$\text{number of level i nodes} = \sum_{k=1}^{\text{(number of level i-1 nodes)}} 1 + \text{N\_keys (First\_prev+k-1)}$$

where First_prev is the node index of the first node at level i-1. That is, First_prev is one greater than the sum of the number of nodes at all levels before i-1. There are more nodes than keys because a node with n keys has n+1 children.

To search a B-tree, is is necessary to be able to locate the child node for a given key, say, key j in node i (i.e., B_tree(i,j)). In the previous definition of a B-tree, the child node is pointed to by the pointer Aj. In this model, with no pointers, the child node's node index will be computed dynamically.

Assuming that the current node i (containing B_tree(i,j)) is at level y in the tree, the node index of the desired node (the jth child of node i) is the sum of the number of nodes at all levels through y plus the number of children of all nodes at level y before node i plus the children of the first j keys in node i. That is:

N1	for the root (1).
+ N2	for the level 2 nodes
+ N3	for the level 3 nodes
+ ...	
+ Ny	for the level y nodes
+ N_keys(N1+...+N(y-1)+1)+1	for the children of the first level y node
+ ...	
+ N_keys(i-1)+1	for the children of the last level y node before node i
+ j	to get to the j-th child node of node i.

But:

$$N1 = 1$$

$$N2 = \sum_{k=1}^{N1} (N\_keys(k) + 1)$$

$$N3 = \sum_{k=N1+1}^{N1+N2} (N\_keys(k) + 1)$$

$$N4 = \sum_{k=N1+N2+1}^{N1+N2+N3} (N\_keys(k) + 1)$$

etc.

So N1 + N2 + N3 + ... + Ny is actually:

$$1 + \sum_{k=1}^{(N1+N2+N3 \ldots + N(y-1))} (N\_keys(k) + 1)$$

or, one plus the sum of N_keys(k)+1 for all nodes in levels 1 through y-1.

Since the level y nodes immediately follow the last level y-1 node in the array, the sum of the number of children of nodes at level y before node i extends the summation to k=i-1. Then, the term j for the children of the j keys up to B_tree(i,j) makes the total become, in the *Child Node Index Equation*:

$$child\_node\_index = 1 + j + \sum_{k=1}^{i-1} (1 + N\_keys(k))$$

or

$$child\_node\_index = i + j + \sum_{k=1}^{i-1} (N\_keys(k))$$

The Child Node Index Equation gives the row index in the ragged array that represents the node associated with the key in B_tree(i,j).

In an implementation using pointers, this is the node that would be pointed to by the pointer associated with that key. Since the node index corresponds exactly to the row index in the ragged array, B_tree(child_node_index,1) through B_tree(child_node_index,N_keys(child_node_index)) is the desired node.

To reference the last child node of node i, which contains keys greater than B_tree(i,N_keys(i)), N_keys(i)+1 is used for j in the above equation. The B-tree search algorithm uses the Child Node Index Equation to traverse a B-tree stored in a ragged array.

3. B-TREE ALGORITHMS

There are four basic B-tree operations that will be considered: searching, inserting, deleting, and scanning.

The implementation described in this paper was done in PL/I on a Data General C350 Eclipse computer. All of the algorithm descriptions that follow are simply those procedures, with variable declarations removed in order to concentrate on the procedural aspects.

There are several procedures used by the B-tree algorithms to perform ragged array manipulation. These procedures will be left unspecified as their form is a function of the ragged array implementation, which is the subject of a later section. For now, the following procedures are assumed to be available:

1) CHILD_NODE_INDEX(NODE_INDEX,KEY_INDEX): returns the result of evaluating the Child Node Index Equation.

2) INSERT_DATA(DATA_INDEX,NEW_DATA): inserts new data into the DATA array as a new record number DATA_INDEX.

3) INSERT_KEY(NODE_INDEX,KEY_INDEX,NEW_KEY): inserts NEW_KEY into the B-tree as B_TREE(NODE_INDEX,KEY_INDEX).

4) DELETE_DATA(DATA_INDEX): deletes the element at DATA(DATA_INDEX)

5) DELETE_KEY(NODE_INDEX,KEY_INDEX): deletes the key at B_TREE(NODE_INDEX,KEY_INDEX). If the last key is deleted from the root node, then the node is deleted and N_tree_levels is decremented.

6) REPLACE_DATA(DATA_INDEX,NEW_DATA): replaces the data in record number DATA_INDEX with NEW_DATA.

7) NEW_ROOT(NEW_KEY): creates a new root creates a new root node (node 1) with one key (NEW_KEY). N_tree_levels is decremented.

8) SPLIT_NODE(NODE_INDEX,KEY_INDEX): splits a node into two nodes by making keys KEY_INDEX+1 thru N_KEYS(NODE_INDEX) into a new node NODE_INDEX+1 and node NODE_INDEX to contain keys 1 thru KEY_INDEX-1. Key KEY_INDEX is deleted. Notice that the two new nodes are adjacent rows in the ragged array.

9) CONCAT_NODES(NODE_INDEX,NEW_KEY): concatenates node NODE_INDEX and node NODE_INDEX+1 together into a new node NODE_INDEX and inserts NEW_KEY between them. The old node NODE_INDEX+1 no longer exists. Notice that the two nodes that are to be concatenated are adjacent in the ragged array.

3.1 Searching

The search algorithm shown in Figure 3 is a modification of the B-tree search algorithm given in [4]. It returns an indication of whether the search key was found, the index of the data associated with the search key, arrays tracing the progress of the search giving the node and key indices of the path through the tree, and the level where the key was found. This last information will be used by the insertion and deletion algorithms. If the search key is not found, the arrays show where the key should be inserted.

```
SEARCH_B_TREE:PROC(START_NODE,START_KEY) RECURSIVE;
   /* Search a B-tree from node START_NODE, key START_KEY                    * /
      DATA_INDEX = START_NODE;
      NODE_INDEX = START_NODE;
      KEY_INDEX  = START_KEY;
      LEVEL = LEVEL + 1;
      IF LEVEL > N_TREE_LEVELS THEN RETURN; /* Bottom of tree */
      IF FOUND    /* If already found */
      THEN DO;  /* All keys in this node are < SEARCH_KEY,            */
              /* so skip to the last key.                       */
         KEY_COUNT = N_KEYS(START_NODE);
         KEY = B_TREE(START_NODE,KEY_COUNT);
         KEY_INDEX = KEY_COUNT; /* Last key                        */
         END;
      ELSE DO; /* Start with the passed in parameters */
         KEY = B_TREE(START_NODE,START_KEY)
         KEY_COUNT = N_KEYS(START_NODE);
         END;
      DO WHILE (TRUE);  /* DO Forever */
         IF KEY > SEARCH_KEY
         THEN DO; /* Found the correct sub-tree */
         NODE_TRACE(LEVEL) = NODE_INDEX;
         KEY_TRACE(LEVEL) = KEY_INDEX;
         IF KEY = SEARCH_KEY
         THEN DO;  /* Also found the key */
               FOUND = TRUE;
               LEVEL_FOUND = LEVEL;
```

```
                END;
            /* Move to the child of this key (EQN 1) */
            NODE_INDEX CHILD_NODE_INDEX(NODE_INDEX,KEY_INDEX);
            KEY_INDEX = 1;
            CALL SEARCH_B_TREE(NODE_INDEX,KEY_INDEX);
            RETURN;
            END;
        IF KEY_INDEX > KEY_COUNT /* No more keys in node? */
        THEN DO;  /* Take the right most sub-tree    */
            NODE_TRACE(LEVEL) = NODE_INDEX;   /* Save the path */
            KEY_TRACE(LEVEL) = KEY_INDEX + 1;
            /*  Move to the last sub-tree (1 + EQN (1)) */
            NODE_INDEX = CHILD_NODE_INDEX(NODE_INDEX+1,KEY_INDEX);
                KEY_INDEX = 1;
            CALL SEARCH_B_TREE(NODE_INDEX,KEY_INDEX);
            RETURN;
            END;
        KEY_INDEX = KEY_INDEX + 1; /* Move to next key in node*/
        KEY = B_TREE(NODE_INDEX,KEY_INDEX);
        END;
        END;
```

Figure 3: B-tree Search Algorithm

There are two parameters and seven global values used by the search algorithm. The parameters are:

1) START_NODE:
   the node index for first key to be compared. The initial call will be with START_NODE set to 1.

2) START_KEY:
   the key index for first key to be compared. The initial call will be with START_KEY set to 1.

The globals are:

1) LEVEL:
the level of the current B-tree node (always set to zero by the caller before the search starts). Used to detect the end of the search.

2) SEARCH_KEY:
the key to be used in the search.

3) FOUND:
a boolean variable, set to FALSE before the search begins. It will be set to TRUE if the key is found.

4) LEVEL_FOUND:
the level of the tree where the key was found, if it is in the tree.

5) DATA_INDEX:
the node index of the object node which contains the data associated with the key (if it is found in the tree). If the object nodes are considered a part of the tree, this value can be used unchanged. Since the objects are stored in a separate array, the correct value for the data index is DATA_INDEX-N_NODES.

6) NODE_TRACE(i):
the node indexes on a path from the root to the lowest level B-tree node (i = 1 to N_TREE_LEVELS). If the key is not found, NODE_TRACE(N_tree_levels) is the row index where the key would be inserted.

7) KEY_TRACE(i):
the indexes of the keys in the nodes in NODE_TRACE on the path from the root to the lowest level of the tree. If the key was not found, it would be inserted as key number KEY_TRACE(N_tree_levels).

The two global arrays are used to remember the path through the tree. These are for the use of the insertion and deletion algorithms, described later.

The caller initiates the search by setting SEARCH_KEY to the desired key value, LEVEL to 0, START_NODE and START_KEY to 1, FOUND to FALSE and then by the reference:

```
CALL SEARCH_B_TREE (START_NODE,START_KEY);
```

The input arguments to the procedure specify that the search is to start with the first key in the root node (B_TREE(1,1)). If the key was found, then FOUND=TRUE and the data index required is DATA_INDEX-N_NODES. The arrays NODE_TRACE and KEY_TRACE contain the trace, and LEVEL_FOUND is the level in the B-tree where the key was found (if it was found). Notice that the procedure is recursive. This is not required, but it simplifies the algorithm.

3.2 Insertion

The insertion algorithm (Figure 4) uses the search procedure to find the place where the key and data should be inserted. The algorithm is an adaptation of the algorithm found in [6].

```
INSERT_B_TREE:PROC(NEW_KEY,NEW_DATA);
  SEARCH_KEY = NEW_KEY;
  FOUND = FALSE;
  LEVEL = 0;
  CALL SEARCH_B_TREE(1,1);
  DATA_INDEX = DATA_INDEX - N_NODES;
  IF FOUND              /* If the key was found */
  THEN DO;              /* just replace the data */
     CALL REPLACE_DATA(DATA_INDEX,NEW_DATA);
```

```
        RETURN;
        END;
    /*  The key was not found, so insert the new data                    * /
    /*  and the new key.                                                 * /
    CALL INSERT_DATA(DATA_INDEX,NEW_DATA);
    LEVEL = N_TREE_LEVELS;
    DO WHILE (LEVEL > 0);
        I = NODE_TRACE(LEVEL);
        J = KEY_TRACE(LEVEL);
        /* Insert the new key as B_TREE(I,J)   */
        CALL INSERT_KEY(I,J,SEARCH_KEY);
        N = N_KEYS(I);           /* Number of keys in node I */
    IF N < M THEN RETURN;  /* Not full, so done        */
    /*   Save the center key    */
    SEARCH_KEY = B_TREE(I,CEIL(N/2));
    /*   Split the node at the center key */
    CALL SPLIT_NODE(I,CEIL(N/2));
    LEVEL = LEVEL - 1;
    END;
    /*   If we get here, a new root node is needed    */
    CALL NEW_ROOT(SEARCH_KEY);

    END;
```

Figure 4: B-tree Insertion Algorithm

3.3 Deletion

The deletion algorithm (Figure 5) is also an adaptation of an algorithm found in [6]. The search procedure is used to find the key and its data.

```
DELETE_B_TREE:PROC(KEY);
   /*  First, find the key*/
   FOUND = FALSE;
   LEVEL = 0;
   SEARCH_KEY = KEY;
   CALL SEARCH_B_TREE(1,1);
   IF ~FOUND THEN RETURN;  /* No key to delete.      */

   /*  Delete the data associated with the key */
   DATA_INDEX = DATA_INDEX - N_NODES;
   CALL DELETE_DATA(DATA_INDEX);

   /*  And delete the key */
   /*  If the key is not in a leaf node */
   IF LEVEL_FOUND < N_TREE_LEVELS
   THEN DO;         /* Replace it with a leaf node key */
                 /* and delete that key in the leaf */
      I = NODE_TRACE(N_TREE_LEVELS);
      J = KEY_TRACE(N_TREE_LEVELS) - 1;
      CALL REPLACE_KEY(NODE_TRACE(LEVEL_FOUND),
                             KEY_TRACE(LEVEL_FOUND),B_TREE(I,J));
      END;
   ELSE DO;   /* The key is a leaf node */
      I = NODE_TRACE(N_TREE_LEVELS);
      J = KEY_TRACE(N_TREE_LEVELS);
      END;
   /* Delete the key at B_TREE(I,J) (Always a leaf) */
   CALL DELETE_KEY(I,J);

   /* If there are too few keys in node I (less than                */
   /* [m/2], we will combine some sibling nodes                     */
   /* to create bigger nodes                                        */

   LEVEL = N_TREE_LEVELS;
   DO WHILE (N_KEYS(I) < CEIL(M/2) - 1 AND I > 1);
      /* There are less than [m/2] keys in node I.      */
```

```
O = NODE_TRACE(LEVEL-1); /* Parent node of node I */
P = KEY_TRACE(LEVEL-1);  /* Parent key of node I   */
IF P < N_KEYS(O)
THEN DO;    /* There is a right sibling */
    K = I + 1;    /* Node K is the right sibling */
    IF N_KEYS(K) > CEIL(M/2)   /* If more than half full */
    THEN DO; /* A key can be deleted from this sibling. */
        /* Move B_TREE(O,P) to the end of node I   */
        /* and move up the first sibling key.       */
        CALL INSERT_KEY(I,N_KEYS(I)+1,B_TREE(O,P));
        CALL REPLACE_KEY(O,P,B_TREE(K,1));
        CALL DELETE_KEY(K,1);
        RETURN;
        END;
    /* Let node I be NODE I || B_TREE(O,P) || NODE K */
    /* and delete node K.                         */
    CALL CONCAT_NODES(I,B_TREE(O,P));
    CALL DELETE_KEY(O,P);  /* KEY(O,P) is now in node I */
    I = O;
    END;
ELSE DO;  /* Must use the Left sibling      */
    K = I - 1;    /* Node K is the left sibling */
    P = P - 1;    /* P is the Parent Key of node K */
    IF N_KEYS(K) > CEIL(M/2)   /* If more than half full */
    THEN DO; /* A key can be deleted from this sibling */
        /* Move B_TREE(O,P) to the beginning of node I   */
        /* and move up the last sibling key              */
        CALL INSERT_KEY(I,1,B_TREE(O,P));
        CALL REPLACE_KEY(O,P,B_TREE(K,N_KEYS(K)));
        CALL DELETE_KEY(K,N_KEYS(K));
        RETURN;
        END;
    /* Let node K be NODE K || B_TREE(O,P) || NODE I */
    /* and delete node I                          */
    CALL CONCAT_NODES(K,B_TREE(O,P));
    CALL DELETE_KEY(O,P);
```

```
            I = 0;
            END;
        LEVEL = LEVEL - 1;  /* Move up a level in the tree */
        END;
    END;
```

Figure 5: B-tree Deletion Algorithm

3.4 Key Order Data Scan

Using the preceding algorithms and B-tree representation, one can see that the DATA array contains the data in key order. That is:

Key for DATA (i) < Key for DATA (j) if and only if i<j.

So a key order scan of the data is simply an index order scan of DATA.

4. DYNAMIC ADDRESS COMPUTATION

The previous section presented algorithms for manipulating a B-tree represented as a ragged array with a small amount of auxiliary information. Several operations are required that are not easily performed on a conventional array:

1) Split node i into two nodes, i and i+1. This means every old node j, j>i is now node j+1.

2) Insert a new node i. Now every node j, j>i becomes node j+1.

3) Delete node i. Now every node j, j>i becomes node j-1.

4) Delete and insert data in the DATA array, with index changes similar to those for nodes, above.

5) Delete and insert keys in nodes, again with key index changes as above.

6) Concatenate two nodes (i and i+1) into one node. This results in every node k>i+1 becoming node k-1.

Using conventional ragged array organization to perform these operations requires either movement of data above the insertion or deletion point, or storage and manipulation of pointers. Cook's Dynamic Address Computation mechanism allows the representation of arbitrarily ragged arrays using no pointers [4], and it performs the above operations with only a small amount of data movement. All of the auxiliary structural information (number of keys in a node and number of nodes) is stored and used by the Dynamic Address Computation algorithms. Finally, some of the information needed by SEARCH_B_TREE, specifically, the sum of the number of keys in nodes before the current node, is generated by the addressing mechanism and can be used by the B-tree algorithms without additional computation.

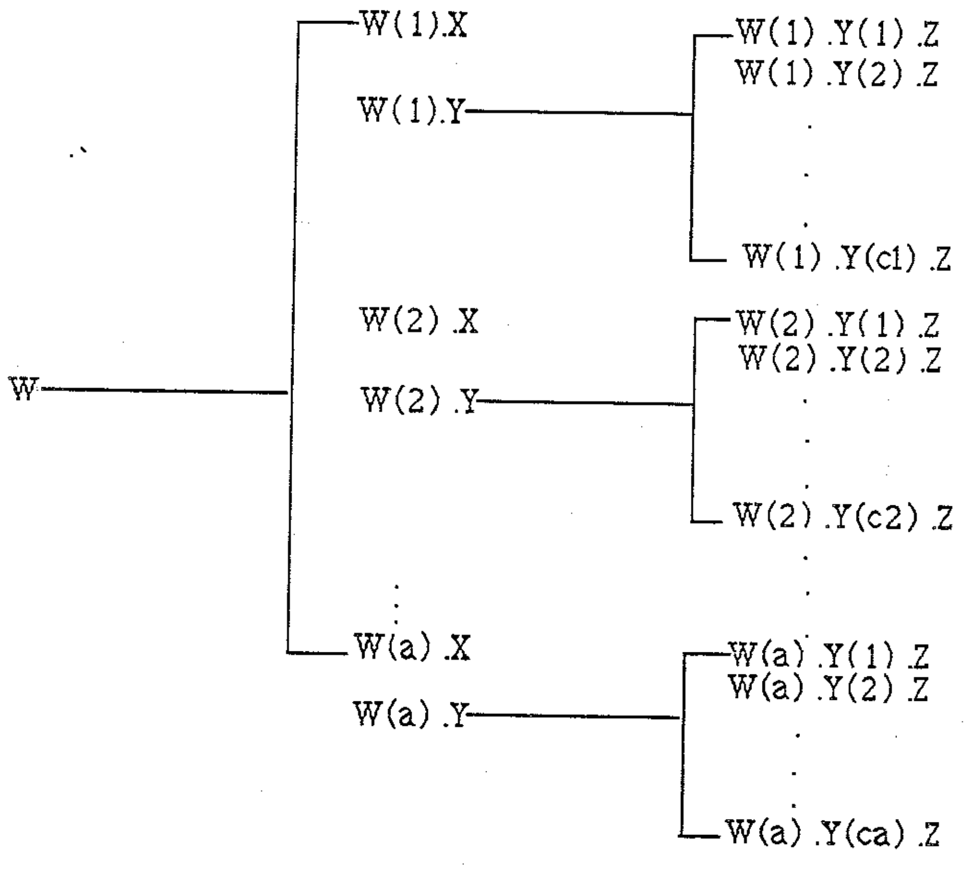
4.1 The Dynamic Address Computation Storage Structure

Dynamic Address Computation (DAC) uses separately stored descriptive data to manage arbitrary structures. Here, structure is being used in the PL/1 sense of a hierarchically ordered set of data. (Cook uses the term "tree", but in keeping with a PL/1-like description of the algorithms and to avoid confusion with the earlier definitions of a tree, "structure" will be used here.) The example in Figure 6, taken from [4], shows the definition of such a structure.

```
01 W            repeats (max A),
   02 X         length (max B),
   02 Y         repeats (max C),
      03 Z      length (max D).
```

Figure 6: DAC Definition of the Structure W

This is the definition of a structure W. Figure 7 illustrates an instance of the structure. In it, X and Z are the data holding components (called LEAF components), with maximum lengths of B and D bits, respectively. W and Y are structural components (called repeating, or REP components) that serve to group the components below them into substructures (Y is called a substructure). There is one occurrence of the structure W, composed of up to A occurrences of the data item X and the substructure Y. Each Y is a (sub)structure composed of up to C occurrences of Z.



where a ≤ A and ci ≤ C.

Figure 7: Populated Occurrence of W

In this example, the sizes of each of the components are described using the syntactic construct "repeats(maxP)" or "length(maxQ)." This means that the components are variable sized, i.e., from 0 to P occurrences of a REP (sub)structure or 0 to Q bits in a LEAF component occurrence. These are called unfactored (UNF) components, since the size of each occurrence (because it is variable) cannot be "factored out" into a single value that describes all such occurrences. A description of the form "repeats(P)" or "length(Q)" would describe a factored (FACT), or fixed size component. In this case, every (sub)structure occurrence would be composed of exactly P occurrences of its descendant REP and LEAF components and every LEAF occurrence would be exactly Q bits long. Now the lengths of each occurrence can be factored out into a single value, thus the name "factored."

In a typical programming language, space would be allocated for exactly A occurrences of X and exactly A*C occurrences of Z, with each occurrence being given the maximum number (B or D) bits. DAC allows the allocation of only as much space as is needed for the actual number of occurrences of X and Y. In addition, each occurrence of Y may have a different number of occurrences of Z, and all X's and Z's may be of different lengths. The case of fixed allocation of space for fixed size data is actually a special case in which all components are factored.

In the a populated occurrence of W shown in Figure 7, notice that there are $a < A$ occurrences of X and Y and:

Y(1) consists of $c1 \leq C$ occurrences of Z
Y(2) consists of $c2 \leq C$ occurrences of Z

.

.

.

Y(a) consists of $ca \leq C$ occurrences of Z

Not shown is the fact that W(i).X is bi bits long for all i and that W(i).Y(j).Z is d(i,j) bits long for all i,j.

The a, bi, ci, and d(i,j) are all tags which represent the number of (sub)structure occurrences and the length of each of the data items. Figure 8 shows the collection of tags for the population shown in Figure 7.

```
Structure
Components    Tags

01  W          a
    02 X       b1,b2, ... ,ba
    02 Y       c1,c2, ... ,ca
        03 Z   d(1,1), ... ,d(1,c1), ... ,d(a,1), ... ,d(a,ca)
```

Figure 8: Tags for Instance of W

The information describing the structure W should require much less space that W itself. This information is all stored in DAC control structures. These structures contain several different groups of information:

1) Descriptors that describe the static organization of the structure (the Schema). Contained here is such information as the number of components and the type of structure (DAC data or user data). The following information is stored for each component:

   a) TYPE:
      The type of component (REP or LEAF).

   b) FORMAT:
      The format of the component (FACT or UNF).

c) TAG_SIZE:

For unfactored components, the size (in bits) of the tags (The ai, bi, ci, and d(i,j)). The tags for one component are all the same size, but different components can have different sized tags.

d) TAG_VALUE:

For factored components, the value of the (single, factored out) tag.

e) TAG_UNITS:

The "units" for a tag. The tag actually stored is reduced by this factor. This reduces the space requirement. The units are used when the data described by the tag is always a multiple of some number of bits long. For example, character data is always a multiple of 8 bits long.

f) PARENT:

An identification of the parent (REP) component of the component being described.

g) EXTENT:

The identifier of the last descendant of a REP component.

2) Page tables, describing where the tags are located.

3) Page tables, describing where the data is located.

The last two tables make up a data structure called the directory, and will be discussed in detail later in this section.

The tags, descriptors and page tables together completely describe the populated structure. Notice that the number of tags for a given (UNF) component (REP or LEAF) is described by the tags in its parent

(REP) component. For example, there are exactly "a" tags for the LEAF component X and the REP component Y. The number of tags for the LEAF component Z is c1 + c2 + ... + ca. Also, the number of occurrences of Z in the i-th occurrence of Y is ci.

The data itself is stored separately. It is stored in a linear fashion, in the same order as it would be encountered in a "pre-order" traversal of the structure. Figure 9 shows the order of the data for the populated structure.

```
W(1).X
      W(1).Y(1).Z
      W(1).Y(2).Z
      .
      .
      .
      W(1).Y(c1).Z
W(2).X
      W(2).Y(1).Z
      W(2).Y(2).Z
      .
      .
      .
      W(2).Y(c2).Z
.
.
.
W(a).X
W(a).Y(1).Z
      .
      .
      .
      W(a).Y(ca).Z
```

Figure 9: Data Storage

4.2 Locating Data Stored in a DAC Structure.

A "reference" to a data item (read, write, insert, delete) consists of one or more component names and associated indices. Thus, W(i).X references the i-th X, W(i).Y references the i-th "substructure" Y, and W(i).Y(j).Z references the j-th Z in the i-th Y. The "address" of a datum is the distance (in bits) from the beginning of the structure to the beginning of the datum.

Notice that, in Figure 7, W(3).X refers to a single leaf occurrence (the third X). Determining the location of W(3).X is a matter of determining the sizes of W(1).X, W(1).Y, W(2).X and W(2).Y. W(3).X follows immediately after the end of W(2).Y. In general, finding W(I).X requires that I-1 occurrences of X and Y be skipped.

The reference W(3).Y is the same, with the important difference that the length of W(3).X must also be considered. That is, to find the third Y, two occurrences of Y and three occurrences of X must be considered. In effect, the structure W has been split between its X and Y components. Finding the referenced component (W(I).Y) requires that I occurrences of the components before the split (X) and I-1 occurrences of components after the split (Y) be skipped.

Notice that the substructure split occurs at the last component mentioned in the reference. Thus W(3).Y requires the split between W(3).X and W(3).Y.

Another example is W(3).Y(4).Z. In this case, the third occurrence of Y is split into W(3).Y(1) through W(3).Y(3) and W(3).Y(4) through W(3).Y(C3). This type of split is not signalled by the position of the component in the structure definition compared to the referenced component, as was the structure split above. Rather, it is simply the presence of an index value for a component. Thus, for W(3).Y(4).Z, W is indexed, so 3 is considered to be an index into the first occurrence of W, and the Y index (4) is considered to be an index into the third occurrence of Y. Thus, two complete occurrences of Y plus three occurrences of X plus 3 additional occurrences of Z are considered.

4.3 The Address of Data

The address of a datum is the sum of the lengths of all data before it in the structure. Referring to figures 8 and 9, the address of W(2).Y(3).Z is:

$$b1 + d(1,1) + \dots + d(1,c1) + b2 + d(2,1) + d(2,2)$$

The bi terms represent the contribution of the X value occurrences to the total distance from the beginning of the structure to the datum, and the d(i,j) terms represent the contribution of the Z value occurrences

In general, the address of Z(I,J) is:

$$\sum_{i=1}^{I} bi + \sum_{i=1}^{I-1} \Big(\sum_{j=1}^{ci} d(i,j)\Big) + \sum_{j=1}^{J-1} d(I,j)$$

Notice that only the tags for the LEAF components are added. The tags for the REP components are used only as limits for the summation. This reflects the fact that only the LEAF components actually hold data.

Computing the address of a particular datum becomes a matter of adding up the lengths of all LEAF substructure occurrences before the desired datum. The REP components are used to determine how many LEAF component tags must be summed.

Cook calls the contribution of a particular component (REP or LEAF) to the address of a datum the DATASPAN for that component. The DATASPAN for a component is the sum of the tags for all occurrences of that component before the referenced datum. For a REP component, the DATASPAN is the sum of the number of occurrences of the substructures that make up that component, or the total number of complete substructure occurrences before the referenced datum. For a LEAF component, the DATASPAN represents the sum of the lengths of all occurrences of that component before the referenced datum, or the total contribution of that component to the address of the datum.

The number of tags that are summed for a given component is called the TAGSPAN for that component. The TAGSPAN for a component can be determined from the DATASPAN of the parent (REP) component or from the DATASPAN of the same component. The DATASPAN for a component can be determined by adding up the first TAGSPAN tags for that component or from the TAGSPAN of a descendent component.

The relationship between the TAGSPAN and the DATASPAN for a component is formally represented in the "Instance Equation".

4.4 The Instance Equation.

For notational simplicity, each component can be referred to by its component number in the structure definition, with the first being 1, the second 2, and so on (For example, in Figure 6, W is component 1 and Z is component 4). Now, the *Instance Equation* for component k is:

$$\text{DATASPAN}(k) = \sum_{i=1}^{\text{TAGSPAN}(k)} \text{TAG}(k,i)$$

Referring to Figure 8, assume that DATASPAN(3) (the DATASPAN for Y) has been computed. This DATASPAN value is the sum of some number of ci. Since ci represents the number of occurrences of Z in the i-th occurrence of Y, this DATASPAN value for Y is the number of Z tags (d(i,j)) that must be summed to get the actual length of DATASPAN(3) occurrences of Z. That is, the DATASPAN for Y becomes the TAGSPAN for Z. Then, the DATASPAN for Z can be computed by adding up the first DATASPAN(4) tags for the Z component.

The inverse is also true. If the TAGSPAN for component Z is known, it can be used as the DATASPAN for its parent, Y. Then, the Instance Equation can be solved for TAGSPAN(3) by counting the number of ci tags that must be subtracted from DATASPAN(3) to reduce it to zero, or below.

The DAC process uses an "address table", contained in the schema and illustrated in Figure 10 for the example structure. Two columns have been added, in addition to TAGSPAN and DATASPAN. INDEX is the index values associated with the data request. For example, the 2 and 3 in W(2).Y(3).Z. The DONE column is used by the address computation algorithm to help control the process.

	TAGSPAN	DATASPAN	INDEX	DONE
W				
X				
Y				
Z				

Figure 10: Example Address Table

The address table is initialized to reflect the form and content of the reference. For the request W(2), Y(3), Z, the address table and two control variables (TOP and BOTTOM) are set as follows:

1) TOP is set to the component number of W.

2) BOTTOM is set to the component number of Z.

3) Only the last component in the reference may be of type LEAF.

4) For each component in the reference that has an associated index value, the INDEX for that component number is set to the index value. The INDEX for all other components is set to 0. If the last component is a LEAF, and has an index value, that value is assigned to its parent.

5) DONE is set to "not done" for all components.

Thus, a reference to W(2).Y(3).Z results in the address table in Figure 11.

The address computation process consists of filling in the empty spaces in the address table. The algorithm is shown in Figure 12.

	TAGSPAN	DATASPAN	INDEX	DONE
W			2	~done
X			0	~done
Y			3	~done
Z			0	~done

TOP=1 and BOTTOM=4.

Figure 11: Initialized Address Table

```
AC:PROC(COMP_NR,DIR,VALUE) RECURSIVE;
  IF COMP_NR = 0 THEN RETURN;          /* Above top  */
  IF DONE(COMP_NR) = NOT_DONE
     THEN RETURN; /*Already done, so can just return */
  IF DIR = UP
  THEN DO;
     DATASPAN(COMP_NR) = VALUE;
     TAGSPAN(COMP_NR)   = GEN_DIV(COMP_NR,DATASPAN(COMP_NR));
     DONE(COMP_NR)      = GOING_UP;
     CALL AC(PARENT(COMP_NR),UP,TAGSPAN(COMP_NR));
     END;
  ELSE DO;  /* Going Down - VALUE is a new TAGSPAN */
     IF ((COMP_NR > BOTTOM) | ((COMP_NR = BOTTOM) &
         (TYPE(COMP_NR)=LEAF)))
     THEN DO; /* Possible substructure split */
        IF DONE(PARENT(COMP_NR)) = DEC_DOWN
          THEN TAGSPAN(COMP_NR) = VALUE - 1;/* Split here */
           ELSE TAGSPAN(COMP_NR) = VALUE;   /* Already split*/
        DONE(COMP_NR) = DEC_DOWN;   /* Split here or above */
        END;
     ELSE DO; /* Above substructure split */
        TAGSPAN(COMP_NR) = VALUE;
        DONE(COMP_NR)      = GOING_DOWN;
        END;
     IF INDEX(COMP_NR) = 0     /* If there is an index    */
                               /* for this component      */
        THEN DATASPAN(COMP_NR) =
              GEN_MULT(COMP_NR,TAGSPAN(COMP_NR) - 1)
                                          + INDEX(COMP_NR);
        ELSE DATASPAN(COMP_NR) =
              GEN_MULT(COMP_NR,TAGSPAN(COMP_NR));
    END;
 IF TYPE(COMP_NR) = REP  /* If COMP. has descendants          */
 THEN DO;
    DO CMP = 1 TO NR_COMPS;/* Then pass the DATASPAN down */
```

```
            IF PARENT(CMP) = COMP_NR  /* If this COMP's parent        */
               THEN CALL AC(CMP,DOWN,DATASPAN(COMP_NR));
            END;
         END;
      END;
```

Figure 12: The AC algorithm

The algorithm moves either "up" the structure, passing a TAGSPAN value up to be used as a DATASPAN, or "down" the structure, passing a DATASPAN value down to be used as a TAGSPAN. An input parameter (VALUE) represents the TAGSPAN or DATASPAN being passed. The direction (DIR) and the number of the component to be processed (COMP_NR) are also parameters to the algorithm. The address table and the control values TOP and BOTTOM are assumed to be globally available. The algorithm is started by initializing the address table, as previously discussed, and the reference CALLAC(TOP,UP,INDEX(TOP)).

Returning to the sample reference of W(2).Y(3).Z, the address table, TOP and BOTTOM are set as previously described. Then, the AC algorithm starts at component W with VALUE=2. Since the direction of computation is UP, the DATASPAN for component 1 is set to VALUE (2) and TAGSPAN is computed to be 1. The DATASPAN for W is the TAGSPAN for X and Y, and is passed down in VALUE. For X, with no input, this will result in a DATASPAN of b1+b2. For Y, with an input value of 3, the TAGSPAN becomes c1+3. Since Y is the parent of Z, c1+3 is passed down to Z. Here, the (sub)structure split occurs (COMP_NR=BOTTOM and TYPE=LEAF). The TAGSPAN for Z becomes (c1+3)-1=c1+2. The DATASPAN is the sum of c1+2 tags, or d(1,1)+...+d(1,c1)+d(2,1)+d(2,2). The final address table for this computation is shown in Figure 13.

	TAGSPAN	DATASPAN	INDEX	DONE
W	1	2	2	go.up
X	2	b1+b2	0	go.down
Y	2	c1+3	3	go.down
Z	c1+3	d(1,1)+... +d(2,2)	0	dec down

TOP=1 and BOTTOM=4.

Figure 13: Completed Address Table

The distance to Z(2,3) is the sum of the DATASPANs for the LEAF components, or:

$$b1 + b2 + d(1,1) + \ldots + d(2,2)$$

which is the result previously obtained.

4.5 Solving the Instance Equation

The Instance Equation can be solved for either DATASPAN or TAGSPAN, given the other. This process is divided into two functions, GEN_MULT, which solves for DATASPAN, and GEN_DIV, which solves for TAGSPAN. Solving for DATASPAN, given a TAGSPAN, involves simply summing TAGSPAN tag values, which is a generalization of the multiplication process.

Solving for TAGSPAN, given DATASPAN is a generalization of the division process. Again, using a factored component as an example:

$$\text{DATASPAN}(k) = \text{TAGSPAN}(k) * C$$

so

$$\text{TAGSPAN}(k) = \text{DATASPAN}(k) / C.$$

For unfactored components, TAGSPAN is the number of tags that must be subtracted from DATASPAN to reduce it to zero (or just below zero, if the tag value(s) do not "divide" DATASPAN exactly) The remainder is defined as the (positive) amount by which the sum of the tags exceeded DATASPAN (This is slightly different than the usual definition of remainder, but it is more useful for DAC).

4.6 Storing the Data and Tags

For large databases, both the tags and the data described by the tags must be stored on disk. In order to avoid mass movement of data when inserting or deleting, the disk space is divided into a number of (fixed size) blocks. The data is then stored in the blocks in pages, which may be from zero to BLOCK_SIZE bits long. If a page grows larger than the block, the page is split and divided between the old block and a second, newly allocated block. The pages, while not necessarily physically contiguous or sequential, are kept logically contiguous and sequential through the operation of the algorithms and data structures described below.

The tags and the data are stored using the same structures. Since the data storage problem is a special case of the more general problem of tag storage, the discussion will be in terms of the storage of tags.

Referring to Figure 8, all of the tags for a single component are shown together on a line. The collection of tags for an unfactored component is called a tag clique (or just clique). In this example, all of the components are unfactored, so there are four cliques. Except for the first component, there is always more than one tag for a given component.

In general, some tags will be factored. For example, if all X's were exactly B bits long (02 W length (B)), the tags for W would be as in Figure 14.

This structure has three cliques (The first component is considered to be unfactored in this example since the value (a) is not fixed).

The storage technique used allows a clique to be independently divided between more than one page and, at the same time, for each page to contain fragments of more than one clique. A page is divided into several (in this case 3) independently varying fragments of data. An example is shown in Figure 15.

Structure Components	Tags
01 W	a
02 X	B
02 Y	c1,c2, ... ,ca
03 Z	d(1,1), ... ,d(1,c1), ..., d(a,1), ... ,d(a,ca)

Figure 14: Tags for W with Factored X Component

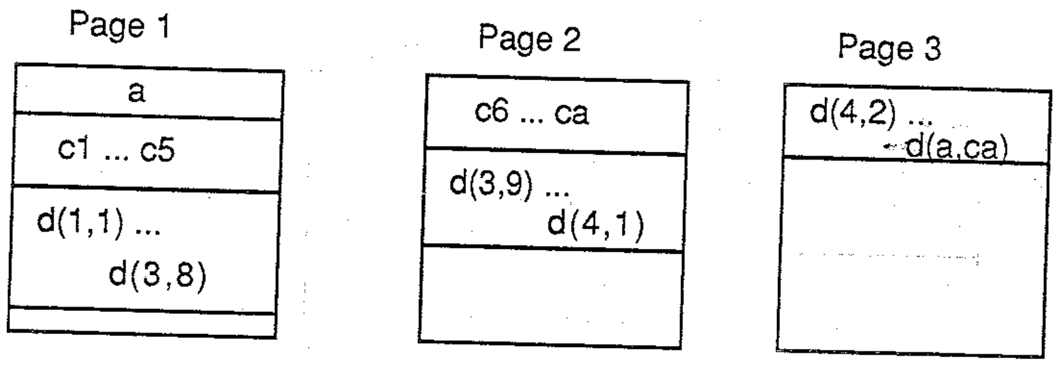


Figure 15: Fragmented Page Storage

Notice that no clique is allocated or stored for X. The tag value (B) is fixed, and is stored in the schema in the descriptor for X.

A method is needed to manage the pages. This involves a new data structure, called a page table. The page table records the amount of data for each clique in each page. There are also algorithms to manipulate the pages and page tables. The page table for the tags (called the tag pages) is shown in Figure 16. It is shown as a DAC structure, as that is how it is implemented.

```
01   TAG_PAGES              repeats  (max  max_pages),
     02   BLOCK_NR          length    (LEN_NR),
     02   PGE_LENGTH        length    (LEN_LEN),
     02  CLIQUES            repeats  (NR_CLIQUES),
          03 CLIQUE_LENG    length    (LEN_LEN);
```

Figure 16: Tag Page Table Structure

BLOCK_NR is the block number on disk where the page is stored. PGE_LENGTH is the total amount of data stored in this page and CLIQUE_LENG is the amount of data for each clique. Thus, if the length of the tags for the components W, Y and Z are w, y, and z, respectively, the pages in Figure 15 would be described by the page table in Figure 17, where the actual values for a, ci, w, y and z must be inserted and the expressions evaluated, since only one number is stored in each position in the table.

	PGE 1	PGE 2	PGE 3
ID.	ID1	ID2	ID3
LENG	w + 5*y + (c1 + c2 + 8)*z	(a-5)*y + (c3 - 8 + 1)*z	(c4 - 1 + c5 + ... + ca)*z
CL_LENG(1)	w	0	0
CL_LENG(2)	5 * y	(a - 5) * y	0
CL_LENG(3)	(c1 + c2 + 8)*z	(c3 - 8 + 1)*z	(c4 - 1 + c5 + ... + ca)*z

Figure 17: Tag Page Table

To reference the N-th tag for component Z (for example) requires first determining the distance, in the third clique, to the tag. This distance is (N-1)*z. A form of generalized divide is now used to determine the correct page. The CLIQUE_LENG(i)'s for the the third clique are added until the sum exceeds (N-1)*z. The number of CLIQUE_LENG entries that must be added determine the page number of the page that contains the tag. BLOCK_NR for this page gives the block number where the page will be found.

For data pages, the concept is identical. The pages are treated as if they were tag pages with only one clique. The page table does not require an array of clique lengths, so the structure in Figure 18 results.

```
01   DATA_PAGES         repeats (max MAX_PAGES),
     02 BLOCK_NR        length   (LEN_NR),
     02 PGE_LENGTH      length   (LEN_LEN),
     02 CLIQUE_LENGTH   length   (LEN_LEN);
```

Figure 18: Data Page Table Structure

4.7 The SITE

A detailed algorithm for mapping a data distance to a page will be presented shortly. First, one last data structure must be discussed, called the SITE.

A SITE is an address, and is the only absolute machine address maintained by the DAC system. It contains the address of a piece of data until the operation on the data is complete, and is discarded when it is no longer valid. The SITE is shown in Figure 19.

```
01    SITE,
      02 BLOCK                      ---,
      02 PGE_OFFSET                 ---,
      02 PGE_RESIDUE                ---,
      02 PGE_REV_RESIDUE            ---,
      02 PGE_INDEX                  ---,
      02 MAP_INDEX                  ---;
```

Figure 19: The SITE

The SITE is a PL/1 structure, and the type declarations for the data components are left unspecified, as they are somewhat implementation dependent. They will all contain integer values, and must be large enough to hold the largest value possible for the implementation.

The SITE holds the address of a piece of data. Referring to Figure 20, PGE_OFFSET is the distance from the beginning of the page to the beginning of the data. PGE_REV_RESIDUE is the distance from the beginning of the clique (in the page) to the data. PGE_RESIDUE is the amount of data in the page following the beginning of the data (in the clique). MAP_INDEX is the page number, and BLOCK is the block number for the page (BLOCK_NR(MAP_INDEX)).

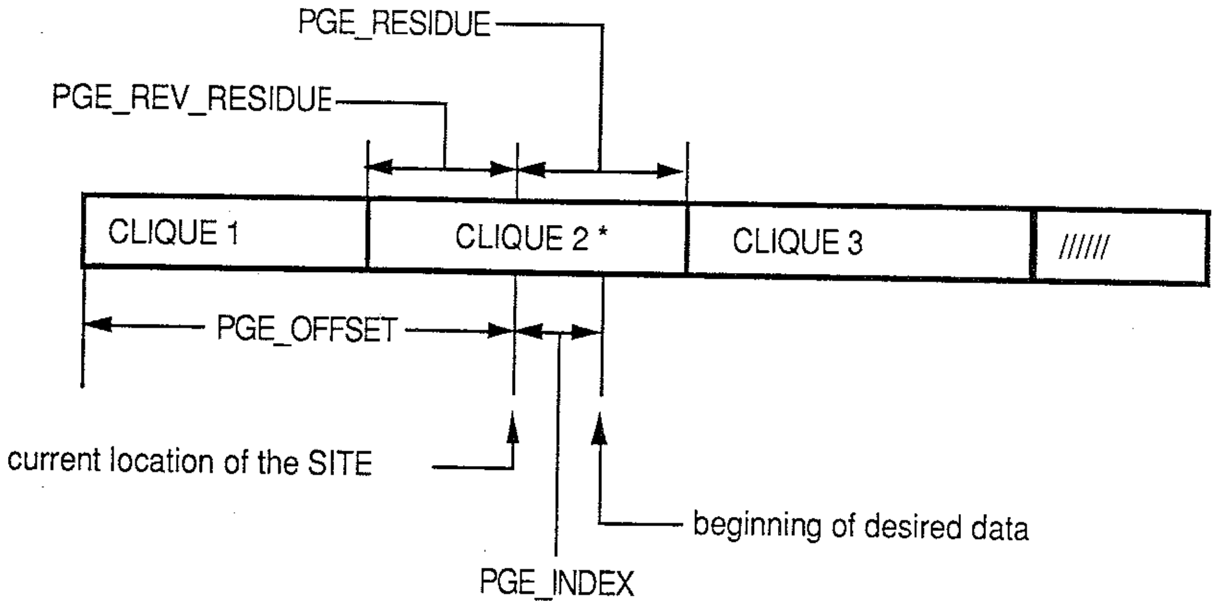


Figure 20: Clique Layout

Thus, MAP_INDEX represents the quotient of the generalized divide operation on the data distance, and PGE_RESIDUE is the remainder. PGE_INDEX is used to indicate a data distance relative to the current location of the site. In the above example, the desired data starts at offset PGE_OFFSET + PGE_INDEX in page number MAP_INDEX which is stored in block number BLOCK_NR on disk.

PGE_INDEX can also be used to refer to data in a different page. In Figure 21, PGE_INDEX is greater than PGE_RESIDUE, so the SITE must be mapped (using the generalized divide) from the current page to a new page. Notice that, while there is still more data in the MAP_INDEX-th page (in the third clique), there is not enough data in that page in the second clique, as the cliques beyond the second one are ignored. This emphasizes the fact that the mapping is done within a single clique, using only PGE_RESIDUE, PGE_REV_RESIDUE, PGE_INDEX, MAP_INDEX and the CLIQUE_LENG entries for that clique.

The CLIQUE_LENG entries for other (lower numbered) cliques enter only into the calculation of PGE_OFFSET.

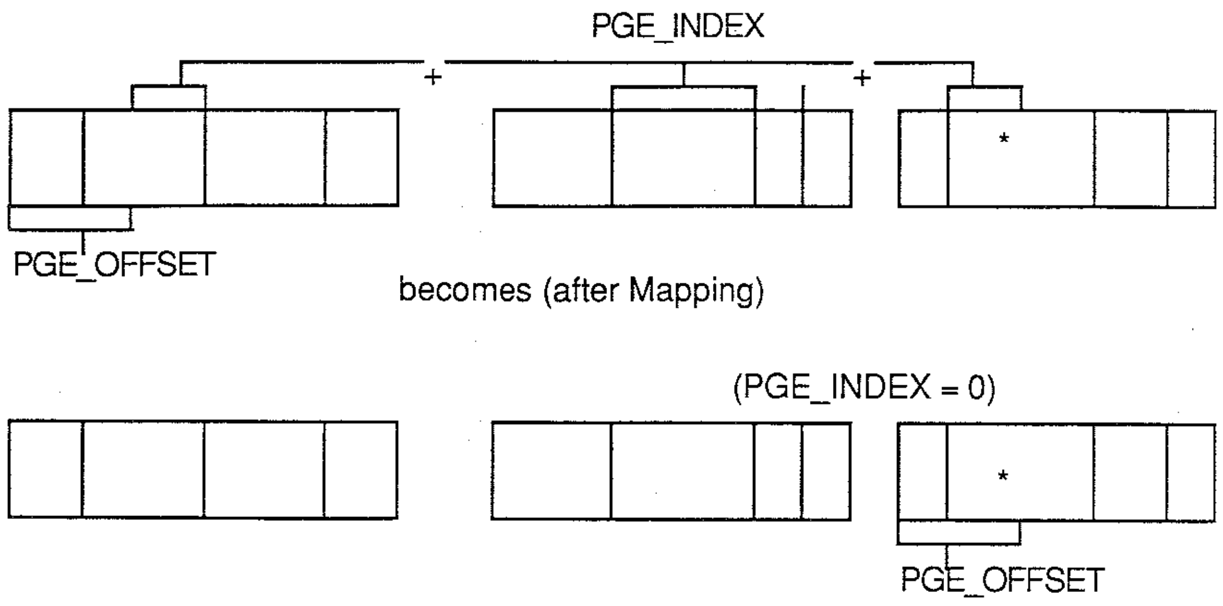


Figure 21: Relationship Between PGE_INDEX and PGE_OFFSET

It is a simple extension to the above concepts to allow the PGE_INDEX to refer to a data distance before the current location of SITE (a backward reference). The site mapping algorithm is called MAP_PAGE and is described in detail in [10].

4.8 Using the SITE and MAP_PAGE to Access Data

Once a SITE has been determined, it can be used to reference the data. This procedure (called GDATA) simple uses MAP_PAGE to re-map the SITE, if necessary, then reads the data from the disk. Similar algorithms are used to insert, replace, and delete data. The details of these procedures are contained in [10].

4.9 The SCHEMA

Several times in the preceding discussion, reference has been made to the schema and directory structures, and pieces have been described. In this and the following section they are presented in detail.

The SCHEMA is a PL/1 data structure that describes a DAC data structure. It has been seen in the previous algorithms as a parameter (STRUC) to the procedures. It contains both the DESCRIPTORS, that describe the components and structure of the DAC data structure, and the ADDRESS_TABLE, used by AC for Dynamic Address computation. There is actually a separate SCHEMA for each DAC structure. The schema is defined in Figure 22 (the actual data types are not specified, as they are implementation dependent).

4.10 The DIRECTORY

The directory is implemented as another DAC structure. This is helpful since the management of the page tables requires that new entries be inserted between existing entries and existing entries may be deleted. The complete definition of the directory is shown in Figure 23.

```
01   SCHEMA,
       02 TYPE_STRUC           ---, /* USER or SYSTEM                      */
       02 DIR_INDEX            ---, /* DIRECTORY Occurrence Number         */
       02 NR_COMPONENTS        ---, /* Number of Components                */
       02 NR_CLIQUES           ---, /* Number of UNFACT Component          */
       02 DESCRIPTORS (NR_COMPONENTS), /* One per Component             */
          03 TYPE              ---, /* REP or LEAF                         */
          03 FORMAT            ---, /* FACT or UNF                         */
          03 TAG_SIZE          ---, /* In Bits                             */
          03 TAG_VALUE         ---, /* TAG = TAG_VALUE for FACT Comp.      */
          03 TAG_UNITS         ---, /* TAG = stored value*TAG_UNITS        */
          03 PARENT            ---, /* Component number, 0 for first       */
```

```
    03 CLIQUE_ID          ---, /* CLIQUE no for UNF Comp only      */
 02 ADDRESS_TABLE (NR_COMPONENTS), /* One per component            */
    03 TAGSPAN            ---,
    03 DATASPAN           ---,
    03 INDEX              ---,
    03 DONE               ---;
```

Figure 22: The SCHEMA Structure

```
01 DIRECTORY                      repeats (max MAX_STRUCTURES),
        02 TAG_PAGES              repeats  (max  max_pages),
            03 BLOCK_NR           length   (LEN_NR),
            03 PGE_LENGTH         length   (LEN_LEN),
            03 CLIQUES            repeats  (NR_CLIQUES),
                04 CLIQUE_LENG length  (LEN_LEN),
        02 DATA_PAGES             repeats  (max  MAX_PAGES),
        03 BLOCK_NR               length   (LEN_NR),
        03 PGE_LENGTH             length   (LEN_LEN),
        03 CLIQUE_LENGTH     length  (LEN_LEN);
```

Figure 23: The DIRECTORY Structure

Each structure described by the directory is given a numeric identifier (DIR_INDEX, contained in the SCHEMA) which is just an index into the DIRECTORY component. Thus, DIRECTORY(N), referring to the entire Nth occurrence of the substructures that make up the directory, describes the Nth structure.

The directory is a DAC structure, and its page tables must be contained in the directory. These tables are contained in DIRECTORY(1) (The first occurrence of TAG_PAGES and DATA_PAGES). Other structures are described by DIRECTORY(2) and higher. This means that the directory's page tables for itself always starts at absolute location 0 in the directory structure. The (known) address of the directory's page tables is contained in a SITE known as the MASTER_SITE.

4.11 Putting it All Together

All of the building blocks required to locate a piece of data have now been described. All that is required is a controlling algorithm to tie them together.

This algorithm is shown in Figure 24. It accepts a user request as a structure identifier and a series of structure components and corresponding indices. It returns a SITE initialized to all zeroes except for PGE_INDEX, which contains the distance from the beginning of the structure to the data.

```
AC_LOC:PROC(STRUC,INPUTS,SITE,DIR_C_SITE,DIR_D_SITE);
 /*                                                              * /
  IF TYPE_STRUC = DIREC_STRUC  /* If a USER structure          */
  THEN DO;            /* Need to locate the directory data    */

     /* Set the DIRECTORY inputs (DIR_INPUTS) to refer            * /
     /* to the first Tag Page Table for the                       * /
     /* DIR_INDEXth occurrence of the DIRECTORY                   * /

     CALL AC_LOC(DIRECTORY,DIR_INPUTS,DIR_C_SITE,
                                MAST_C_SITE,MAST_D_SITE);
     CALL MAP_PAGE(DIRECTORY,DIR_C_SITE,FWD,MAST_D_SITE,1);

     /* Set DIR_INPUTS to refer to the Data Page Tables */

     CALL AC_LOC(DIRECTORY,DIR_INPUTS,DIR_D_SITE,
                                MAST_C_SITE,MAST_D_SITE);
     CALL MAP_PAGE(DIRECTORY,DIR_D_SITE,FWD,MAST_D_SITE,1);
     END;
  ELSE DO; /* DIRECTORY access; use the MASTER SITES */
     DIR_C_SITE = MAST_C_SITE;  /* The tag (CLIQUES) MASTER*/
     DIR_D_SITE = MAST_D_SITE;  /* The DATA MASTER         */
     END;
  END;
/* Now locate the requested data */
CALL AC_INIT(STRUC,INPUTS,TOP,BOTTOM);
CALL AC(STRUC,TOP,UP,INDEX(TOP));/* ADDRESS COMPUTATION */
BLOCK_NR = 0;                       /* and set up the SITE         */
PGE_OFFSET = 0;
PGE_RESIDUE = 0;
PGE_REV_RESIDUE = 0;
PGE_INDEX = 0;
MAP_INDEX = 0;
DO LEV = 1 TO NR_COMPS;
   IF TYPE(LEV) = LEAF
      THEN PGE_INDEX = PGE_INDEX + DATASPAN(LEV);
   END;
END;
```

Figure 24: Locating the Data

Satisfaction of a request for a data location requires that the location of the directory data for that structure be known (this is the DIR_SITE). This is just a request for the location of some data with structure identifier DIRECTORY and with the first directory component being indexed by the numeric identifier for the structure being referenced (DIRECTORY(DIR_INDEX)). This results in a recursive application of the algorithm. Now, the location of the directory's directory data must be determined. This would result in another recursive application of the algorithm, etc. The algorithm will terminate, because the directory's data has been assigned to a known location (first in the directory), and this location is returned (this is the MASTER_SITE).

There are two sets of page tables, one for the tags (the tag page tables) and one for the data (the data page tables). Thus, there are two MASTER_SITEs and there will be two DIR_SITE's set. Since MAP_PAGE expects to be able to do relative addressing using an already mapped DIR_SITE, the PGE_INDEX in this site must be zero. MAP_PAGE is therefore used to "pre-map" the directory sites.

After the directory data is located (for either the directory or the user structure), an address table is initialized using the inputs, and AC is used to perform address computation. The SITE is set to all zeroes, then PGE_INDEX is set to the sum of the DATASPAN values for all LEAF components.

Once the data is located, the SITE is passed to the appropriate routine to read, delete, insert, or replace the data.

5. IMPLEMENTING B-TREES USING DYNAMIC ADDRESS COMPUTATION

In the previous discussion of algorithms for manipulating B-trees, several procedures for performing operations on the underlying ragged array were mentioned, but their details were left unspecified. Also, B_TREE, N_KEYS and DATA were treated as arrays, and N_NODES as a simple variable. They are all implemented as procedures which perform address computation and retrieve or store keys, data or tags. All of these procedures are presented in detail in [10].

These procedures collectively act as an interface between the B-tree algorithms and the dynamic address computation algorithms. Conceptually, they provide the representation of the B-tree as a ragged array.

Using the notation in the previous sections, the array B_TREE for a B-tree of degree m with varying length keys is defined in Figure 25.

Similarly, Figure 26 defines the DATA array for varying length data records.

```
01 NODE                  repeats (max MAXNODES}),
   02 KEYS_PER_NODE      repeats (max m-1),
      03 KEY             length  (max KEY_LENGTH);
```

Figure 25: DAC Definition of the B-TREE Array

```
01 DATA_NODE             repeats (max MAXDATA),
   02 DATA               length  (max DATA_LENGTH):
```

Figure 26: DAC Definition of the DATA Array

Now, to reference keys or data requires only that AC_WC and GDATA be used to locate and read the data. Other calls are used to insert, replace or delete data, or to manipulate the tags, which represent the length of data or the number of keys in a node.

6. PAGE MANAGEMENT AND PERFORMANCE IMPROVEMENT

The preceding sections described the "high level" DAC algorithms. These discussions ignored the low level problem of actually writing and reading blocks to and from a file. It is possible to improve the performance of these algorithms (and, incidentally, to simplify their analysis) by making several modifications. This section briefly describes the method used to manage the transfer of data from disk to memory and the methods for improving the system performance. The two subjects are discussed together because, in some cases, they are related.

6.1 Page Management

The data accessing algorithms previously described (GDATA, and the corresponding algorithms for inserting and deleting data) all assume the ability to reference (read and write) bits in a block on disk given the block identifier (Block number), a page offset (bit position) and a length (in bits). The simplest way to implement this is to simply read the block, extract or replace the correct bits and then (for insertion, deletion and replacement) to write the block back to its previous position on disk.

Normally, there are several references to a block. A small scale version of the classical demand paging concept used in operating systems for virtual storage [2] was used to take advantage of this fact. The buffer to be used is chosen using a simple Least Recently Used (LRU) replacement algorithm.

6.2 Pinning Blocks

There are several modifications to the basic DAC and page management algorithms that improve the overall performance of the system. Some of these were described by Cook in his papers and others are introduced here. Not all of the ideas discussed here were actually implemented, but those that were implemented resulted in notable performance improvements (discussed later). We begin with the concept of pinning blocks.

In the basic LRU replacement scheme, all blocks in memory are subject to being replaced when a new block must be read. In the DAC directory, there are three blocks that are very heavily used. These are the pages that hold the page tables that describe the directory itself, and the page with the tags for the directory. The DAC algorithms require by assumption that the page tables for the directory never exceed one page for the tag page tables and one for the data page table. The tags for the directory do not require very much space. In fact, the next section describes a modified tag organization that will reduce the directory tag space requirement to a very small, fixed amount.

These blocks are needed by virtually every data reference. The tags are required to locate the directory data for the other structures, and the page tables are needed to map the directory sites used to reference the directory data for the user data and tags (the MASTER_SITES). Because the blocks are so often used, it is likely that the LRU mechanism would always keep them in memory. In order to be certain, however, these blocks are "pinned" into memory. That is, they are read once, when the file is opened, into buffers that do not participate in the LRU replacement process. This insures that these blocks are only read once (and written once at closing time, if they are modified).

6.3 Locally Factored Components

An earlier discussion described two component formats, Factored and Unfactored. A third type was also identified by Cook, and can be used to some advantage in this implementation. This new format is called "Locally Factored" (LFACT). A locally factored component is one in which the value of the components tags are not fixed for all occurrences of the substructure or datum it owns (as it is for a factored component), but the values do not change for every occurrence of the substructure (as they do for unfactored components). Instead, the tags for a locally factored component are fixed for all occurrences of the substructure or datum within a single occurrence of the substructure that contains the substructure or datum.

A relevant example is the TAG_PAGES substructure in the DIRECTORY structure [10]. Here, the tags for the CLIQUES component represent the number of CLIQUE_LENG (LEAF) entries in a single tag page table entry (one occurrence of the substructures making up the TAG_PAGES structure). The tag value is the number of cliques in the page described by that entry. Notice that the number of cliques varies between the various data structures described by the DIRECTORY, but that for a single data structure, all of its tag page table entries have the same number of cliques. This means that the CLIQUES tags for all page table entries for a single data structure will be the same. Stated differently, within one occurrence of the TAG_PAGES sub-structure of DIRECTORY, all of the tag values for the CLIQUES component are the same.

Now, notice that the number of CLIQUES tags for a given occurrence of TAG_PAGES is given exactly by the TAG_PAGES tag for that occurrence. A locally factored tag is only stored once, and the DAC algorithms that depend on the tags (GEN_MULT and GEN_DIV) use the value of the parent tag corresponding to the substructure occurrence to determine the number of substructures to which the single tag applies. The parent tag is the number of times that the tag would have occurred, had it been an Unfactored tag.

This results in savings two ways. First, the amount of storage required for the tags is decreased, since these tags are only stored once per occurrence of the containing (sub)structure, instead of once per occurrence of the described substructure. The second savings is in processing time, since fewer tags need to be read when executing the AC algorithms of GEN_MULT and GEN_DIV.

With locally factored tags, the single tag and the parent tag are each read, and the locally factored tag applied as many times as the value of the parent tag. With unfactored tags, the parent tag is not read, but at the component of interest, many tags must be read. In fact, the break-even point in terms of number of tags read occurs when the number of (sub)structure occurrences (and the value of the parent tag) is two.

There is also a savings in update costs since a locally factored tag need only be inserted when a new occurrence of the containing substructure is created. After that, all new occurrences of the described substructure only result in the parent tag being incremented, an update that is also required for unfactored tags in addition to the insertion if the new tag.

Locally factored tags are used in this implementation for the CLIQUES component of the TAG_PAGES substructure of the DIRECTORY data structure. Thus, for the B_TREE structure, its associated DATA structure and the DIRECTORY structure, the total number of CLIQUE tags is fixed at three. The total number of tags for the DIRECTORY structure is therefore fixed, and small (there are three TAG_PAGES tags, three DATA_PAGES tags, one DIRECTORY tag and all other components are factored).

6.4 Restart Points

In the DAC algorithms described so far, after address computation has been performed to locate a datum (filling in an address table), the results are discarded before the next reference is processed. If no updates have occurred, and the second reference is "near" the first (compared to its distance from the beginning of the data structure), the results of the first address computation can be used as a "starting point" for the second. This is accomplished using what Cook calls "restart points". A restart point stores enough information to restore the address table to its state after locating a datum. An additional column is added to store the remainder after a generalized divide. Now, GEN_MULT uses only the difference between an incoming TAGSPAN value (the DATASPAN of the parent) and the TAGSPAN found in the restart point to do the multiplication in computing a new DATASPAN. The REMAINDER is added in to the DATASPAN in addition to any tags that are read. GEN_DIV also uses the difference between the restart point values and the incoming values in its computation. Here, the REMAINDER indicates how much further in the substructure the computation can go before reading (dividing by) another tag. The result is that fewer tags need to be read. Of course, this results in performance improvements only for data structures with unfactored components, since factored components can use the hardware multiply or divide with tags directly from memory (in the DESCRIPTOR).

There are several possible ways to use restart points. One is to have many distributed throughout the data structure. Those would be updated by the DAC algorithms whenever updates occurred. Another possibility is to keep multiple restart points, but to invalidate any that are beyond the location of an update. This requires much less maintenance and is the method used here.

Another issue is the choice of a restart point, and when it should be updated to refer to a different datum. The restart point can be chosen automatically or specified by the user, and it can be left pointing where it was before its use or moved to point to the latest reference. In the implementation described here, the user can decide whether to let the system chose the restart point, to specify a particular restart point or to not use restart points at all. The user also can specify that the restart point is to be updated to refer to the new location, that a different restart point is to be updated or that no point is to be updated. All restart point usage can also be disabled, for performance comparisons.

6.5 First Component Tag

A simple, but useful modification involves the tag for the first component of a data structure. This component is usually thought of as unfactored, since its value can vary, depending on the number of substructure occurrences. However, there is only one tag for this component, so it meets all requirements for being factored except that its value is not fixed. Simple modifications to the tag referencing algorithms allow this tag to be maintained in the

DESCRIPTOR, just as is done for factored components, but to be updated as the data structure evolves, as is done for unfactored components.

This results in advantages for all references to this tag, since it is now permanently in memory, instead of being on disk. It is especially useful for a data structure that is completely unfactored except for the first component (for example, the DATA structure in the B-tree system, when the indexed data is fixed length).

If the tag were kept on disk, as are most unfactored tags, it would reside in some block on disk. References to the data could require reading and writing the block, along with the page table required to map directory sites to the tag. Putting the tag in memory eliminates all of the direct I/O overhead just described, plus any extra I/O caused by the LRU replacement algorithm.

This modification was made to in the implemented system.

6.6 Restart Sites

Later, it will be seen that a significant portion of the disk accesses required to reference a datum are involved in mapping of sites, especially the directory sites used to reference the page tables to map the user data sites. The same reasoning that led to the concept of restart points for address computation leads to the concept of "restart sites". A restart site could be associated with a restart point, or could be independent.

A restart site is a normal site, but with the addition of a TOTAL_OFFSET field that contains the total offset represented by the site (remember that a SITE contains only the offset within a given page). The mapping already done in the site does not have to be redone each time a datum near the restart site is referenced. Instead, the SITE is set to the contents of the restart site, and the PGE_INDEX field set to the difference between the total offset to the datum and the total offset in the restart site. Then, the SITE need only be mapped from its current location to the offset required, instead of being mapped the entire distance from the beginning of the data structure.

Restart sites have to be maintained just like restart points, and any modifications to the data would again invalidate restart sites beyond the point of the modification. This modification was not made in the implemented system.

7. PERFORMANCE ANALYSIS

The analytical evaluation of the performance of the B-tree algorithms when implemented using DAC turns out to be a fairly difficult project. The performance measure chosen (number of blocks transferred to and from disk) is affected by everything from the B-tree algorithms themselves at the highest level to the LRU replacement algorithm at the lowest level. Fortunately, some of the performance improving modifications described in the previous section also make the analysis somewhat more tractable.

Following Wiederhold [9], we consider one of the performance measures to be the (disk) storage required to hold the tree. This analysis is presented in [10]. This section will present an analysis of one other aspect of performance—the cost (in disk blocks transferred) to search the B-tree (and read the indexed data).

The real cost (in time) to read a block can vary widely, even for the same device. It depends on such factors as seek time and rotational latency. It is common to assign "average" values for seek time and latency (see [9]), but, algorithm design and analysis should account for successive reads to blocks that are on the same track or cylinder, thus requiring no seek. In modern multi-programmed computers, requests from many different programs may be made to the same disk drive. This means that there may be little opportunity for an individual program to take advantage of adjacent blocks, or even to control placement of data at all on a disk. For this reason, the cost for a block will be considered to be a constant. If these algorithms were implemented for a dedicated disk, page placement would become a consideration.

Other measures (cost to insert, delete or replace keys and data) will not be derived. This course has been chosen since every reference to data in the tree first requires that a search be performed. This means that all of these other costs will have been partially defined in the search analysis. The cost to actually insert or delete a key has been left for future research.

7.1 Notation

The following notation will be used throughout this section for the basic parameters.

m -	Order of the B-tree.
x -	Number of levels in the tree.
a -	(Average) number of keys in a node.
K -	(Average) length of the Keys.
D -	(Average) length of the Indexed Data.
Nn -	The number of B-tree nodes.
Nk -	Number of keys.
Nd -	Number of Indexed Data entries (Nd = Nk).
Tkpn -	Length of a single KEYS_PER_NODE tag.
Tkey -	Length of a single KEY tag.
Tdata -	Length of a single DATA tag.
Ttp -	Length of a single TAG_PAGES tag.
Tcl -	Length of a single CLIQUES tag.
Tdp -	Length of a single DATA_PAGES tag.
Bk -	Block size.

Ld -	Loading Factor (Bk * Ld = Average data on a page).
LEN_LEN -	The length of a length entry in a Page Table (ex. CLIQUE_LENG).
LEN_NR -	The length of a BLOCK_NR entry in a Page Table.
LASTn -	The Node index of the last node read.
LASTk -	The Key index of the last key read.
LASTo -	The ordinal number of the last key read, in linear order from B_TREE(1,1) => 1, B_TREE(1,2) => 2, etc.

The last 3 values depend very much on the distribution of the keys in the B-tree. For this analysis, we assume that the keys are uniformly distributed across the possible range of key values, and that they are uniformly distributed through the nodes in the B-tree.

7.2 Locating Keys and Tags

The root node contains between 1 and m-1 keys, for an average of m/2 keys. The other nodes contain between (m-1)/2 and m-1 keys, for an average of 3/4(m-1) keys (a).

For an "average" search of the B-tree, we assume that the middle key is read and that this requires reading to the middle key of the middle node at each level. Then, LASTn would be the middle node of the last level. This means that:

$$\text{LASTn} = \sum_{i=1}^{x-1} \left(\text{\# of level i nodes}\right) + \frac{\text{\# of level x nodes}}{2}$$

1. The number of level 1 nodes is 1.

2. The number of level i nodes for $2 \leq i \leq x$ is:

$$\sum_{j=1}^{\text{number of level i-1 nodes}} \left(1+\text{N\_keys}(\text{First\_prev} + j - 1)\right)$$

There is no closed form solution for these equations, so the algorithm in Figure 27 is used.

```
FIRST_PREV  = 1;
NR_AT_PREV  = 1;
LAST_N = 0;
LEVEL = 2;
DO WHILE (LEVEL <= X)
            LAST_N = LAST_N + NR_AT_PREV;
     NR_AT_LEVEL = 0;
     DO I = 1 TO NR_AT_PREV;
          NR_AT_LEVEL = NR_AT_LEVEL+N_KEYS(FIRST_PREV+I-1);
          END;
     FIRST_PREV = FIRST_PREV + NR_AT_PREV;
     NR_AT_PREV = NR_AT_LEVEL;
     LEVEL = LEVEL + 1;
     END;
LAST_N = LAST_N + CEIL(NR_AT_PREV/2);
```

Figure 27: Computation of LASTn

LASTk is simply the middle key of node LASTn, so:

$$\text{LASTk} = \frac{\text{N\_keys (LASTn)}}{2}$$

LASTo is the sum of the number of keys in all nodes in levels above the lowest plus the number of keys in all nodes before the last (LASTn) plus the key index of the key in the last node:

$$\text{LASTo} = \text{LASTk} + \sum_{j=1}^{(\text{LASTn} - 1)} \text{N\_KEYS}(j)$$

$$= \text{the Child Node Index Equation - LASTn}$$

The analysis is for unfactored keys and data since this is the most complex form and the analysis for factored keys and data is a special case. The cost to search the tree can be divided into two parts:

1) The cost to locate and read the B-tree data (Tags and Keys).

2) The cost to locate and read the directory data for the B-tree and its tags.

Recalling the ragged array model of the B-tree, observe that the process of searching the B-tree moves "forward" through the array. That is, the first node read is always node 1. The next node is chosen using the Child Node Index Equation, and always has a node index greater than the node just read. The same is true within a node. The first key in the node is read, then the second, etc. Thus, the displacement to a given key (computed by AC) will always be greater than the displacement to the previously read key, and less than the displacement to the next key to be read.

7.3 Reading Tags

Now, consider only the problem of reading the tags in the AC algorithm. For the purposes of this analysis, we will assume that one restart point is being used to start each Address Computation, and is being reset to the just located datum when the AC algorithm finishes. To read the first key (B_TREE(1,1)), the restart point is not used, but is set when the data has been located. There are only two tags read by AC for this key; the first KEYS_PER_NODE tag and the length tag for the first key. The length tag will be read again to determine the length of the key, but since the block containing this tag was the last one read by AC, the LRU replacement algorithm will result in a "free" reference to the page the second time.

The next key read will use the restart point as its starting place. This means that the tags read by AC when the restart point was set need not be read again, as their contents are already reflected in the values in the address table. In fact, if the second key read is B_TREE(1,2), only a single additional KEY tag must be read. It will be read twice, but neither read will result in a disk access, since the page was read for the previous key (assuming that the two tags are on the same page).

Now, consider reading the tags for the last key to be read (B_TREE(LASTn,LASTk). With the restart point, none of the tags used to locate the previous key are re-read. Only the tags beyond those read for the previous key need be read.

Thus, since the keys are being read in a forward direction, each tag is read only once. The second reading of those tags corresponding to the lengths of the keys read will never result in a page fault, as previously discussed.

Thus, each tag page for the B-tree from the first through the one containing the KEY tag corresponding to the last key read must be read exactly once. With LASTo being the ordinal KEY number of the last key read, the number of pages read is the number of tag pages required to store the first LASTo KEY tags and the first LASTn KEYS_PER_NODE tags:

$$\text{Tag pages read} = \frac{\text{LASTo} * \text{Tkey}}{\text{Ld} * \text{Bk}} + \frac{\text{LASTn} * \text{Tkpn}}{\text{Ld} * \text{Bk}}$$

Once AC has computed the offset to the key and its length, the key itself must be read. The Directory data is used to determine which page holds the key, and only the block for this page is read from disk. This is the last block referenced before control is returned to the caller. If the next key in the same node is read, the only blocks read to locate the key will be (at most) one tag Block (because of the restart point) and a very few Page Table blocks. If this key is on the same page as the previous key, since that block was the last one read before DAC began locating the new key, there must be (Number of buffers) new blocks read if the key data page is to be replaced. For this analysis, we will assume that there are enough buffers available so that the key data page is not replaced.

Thus, the blocks holding only the keys actually referenced must be read exactly once. The expected number of B-tree data pages read to find the last key read is the number of blocks required to store an average of a/2 keys of length K at each of x levels:

$$\text{B-tree Data Pages Read} = \frac{x * (a/2) * K}{\text{Ld} * \text{Bk}}$$

The second cost (locating and reading the Directory data) is somewhat more difficult to predict. Recall that in order to locate the Directory data for the B-tree, the Directory data for DIRECTORY is first located. But, this is free, since this data is stored in the MASTER SITEs, whose locations are always known.

Now, to locate the directory data for the B-tree involves the use of AC on the DIRECTORY data structure. But, the Directories page tables and tags are all in the first three blocks, which have been pinned in memory. So, no matter how many times these pages are referenced, no page faults occur. The only cost left is the cost of mapping sites for tags and data in the B-tree.

For mapping sites for tags, careful consideration of the amount of data that can be stored in a page can simplify the analysis. First, a single block can hold

$$Ntpt = \frac{Ld * Bk}{LEN\_NR + 3 * LEN\_NR}$$

Tag Page Table entries for B-tree Tag Pages. Also, a block can hold

$$Nt = \frac{Ld * Bk}{Tkpn * Nn + Tkey * Nk}$$

tags.

This means that one Tag Page Table page is sufficient Ntpt * Nt tag pages for the B-tree. Both Ntpt and Nt tend to be large, since the tag page table entries and the tags themselves are small. Careful tuning of Block size based on the number of keys can easily result in only one block being required for mapping B-tree tags (in the implemented system, a single page can map approximately 30,000 keys). Thus, to map the tags for the B-tree:

Tag Page Table Pages read = 1

For mapping the data, the cost is much higher since the keys take up much more space than the tags. To locate the page indexed by a site requires reading the PGE_LENGTH entry for every page from the first through the page that contains the offset of interest. Then, the CLIQUE_LENG and BLOCK_NR entries for the page must be read. This mapping is done from scratch for each Key read, since SITE level restart points were not implemented. The largest possible cost is the cost of mapping to the last page (e.g., for the largest key in the tree). This would require that all of the Page Table blocks for the data be read exactly once. A block can hold

$$Ndpt = \frac{Ld * Bk}{LEN\_NR + 2 * LEN\_NR}$$

Data Page Table entries for B-tree Data pages. As for the tags, this number is fairly large, so the number of Page Table pages is a small fraction of the number of Data pages.

Since site relative addressing is used in MAP_PAGE, no Page Table block will ever be read more than once per key. Also, since the page tables are fairly heavily used, as long as data mapped by the entries on the first Data Page Table Page are being used exclusively, the page will probably always be in memory. Conversely, since each page table can describe so many data pages, it is likely that once the reference has moved from one page table page to the next, the first page of the page table will not survive in memory. Thus, as long as keys are being read that can be mapped to the first Page Table page, the single initial read of the page will suffice, but when the later keys are being read, all of the Page Table pages required will probably be read for each key (up to the page containing the Page Table entry for the correct data page).

We are assuming that a node is about the same size as a block. The average length of a key should be less than the size of a B-tree Data Page Table entry (for the implemented system this was 39 bits, or about 5 bytes). So the first Page Table page will generally be able to hold more page table entries than the root node holds keys. The number of keys in the root node is two less than the number of nodes in the first two levels, so we can safely assume that the first page of the Page table maps all of the first two levels. In general, the swapping of page tables will not begin until the tree has grown to more than Ndpt nodes, and will occur only when the search has moved beyond the second level. The worst case would occur when the third level node was at a displacement great enough to require entries from the Page Table pages beyond the first. Then, the number of pages read for a 1 or 2 level B-tree would be 1. For a larger B-tree, the cost is the cost to read an average of 1/2 of the keys in one node at each level after the second. The "average" access to the middle of the data structure requires reading 1/2 of the Data Page Table pages for each key read.

The number of Data Page Tables for the B-Tree is:

$$\frac{\text{Number of B-tree Data Pages}}{\text{Ndpt}}$$

So:

For x = 1 or 2, the number of pages read is 1.

For x > 2, the number of pages read is:

$$(x - 2) - \left(\frac{a}{2}\right)\left(\frac{1}{2}\right)\frac{\text{Number of Data Pages for B-tree}}{\text{Ndpt}}$$

where a = average number of keys in a node.

8. CONCLUSIONS

In this paper, a complete implementation of Dynamic Address Computation (DAC) as described by Cook [4] was presented in detail. The paper describes a working implementation in PL/1 on a 16 bit minicomputer (Data General Eclipse C350). The working implementation was used to measure elapsed time performance of the system on a dedicated computer. Analytical performance measures of disk access cost were derived that can be used to compare this implementation with other, standard implementations. These comparisons are made later in this section, and it is shown that, in terms of performance, more work is required to make the DAC based implementation competitive with the traditional implementations. The DAC mechanisms require hardware support to be viable.

In addition, this section contains some comments about the implementation, lessons learned, and possible future directions.

8.1 Hardware Considerations Make a Difference

The system was implemented on a 16 bit Data General C350 Eclipse processor, using PL/1. No assembly language was used. The first implementation used a data type (Packed Decimal) for the DAC computations and a PL/1 runtime library that caused the data type to be simulated using software. The result was very poor performance. The next version used a hardware supported data type (Double Precision Floating Point) and the performance increased by about a factor of 2. Even then, the program is CPU-bound on the 16 bit mini; approximately 75% of the elapsed time to build a large B-tree is spent in the CPU. This suggests that, for a general purpose computer and an implementation in a high level language, the limiting factor on performance may not be the disk, as originally assumed. Instead, the volume of computation required to do the address computation makes the CPU performance more important.

Cook has suggested that special hardware with (for example) instructions to add up tags could be used to advantage in DAC. Given the current trends in hardware, with CPU costs and speeds improving faster than corresponding factors for mass storage, this is an interesting area to explore. There are also some opportunities for parallel processing in DAC, particularly when adding up tags.

8.2 The Performance Improvements Made a Difference

Some preliminary measurements of CPU and elapsed time on a dedicated system indicated that restart points improved the performance of a B-tree with unfactored keys by a factor of 3. For factored keys, the performance improvement is only about 10%. This is not unexpected, since restart points are not effective for factored components.

The same data indicated that the performance of a B-tree with factored keys was about 4 times that of unfactored keys when restart points were not used. With restart points, the difference between factored and unfactored keys was only a factor of 1.5. This emphasizes the point of the preceding section, that the major portion of the time spent searching the B-tree is consumed in address computation (adding up tags).

8.3 More Work is Needed

Coincidentally, a "standard" B-tree implementation was in progress on the same machine used for the work described here. This implementation was in ALGOL and assembler, and used algorithms as described in [8]. Its performance on a dedicated machine (in terms of keys stored per unit of elapsed time) was an order of magnitude better that the system described here (10 to 20 times for the best DAC cases). Much of the time saving could be accounted for by the low level language and a more careful fit to the operating system parameters. Conversely, it did not seem to suffer too much from lack of node space due to pointers (in fairness, a key compression algorithm was used to reduce the space required to store the keys). The standard implementation also exhibited more robustness in terms of constant performance as the number of keys grew. The DAC based system described here would need a lot of careful work to produce a B-tree manager with performance better than a carefully done standard implementation. Again, this emphasizes the need for hardware and operating system support of DAC to obtain good performance.

The analytical measures lead to a similar conclusion. Applying the analytical performance measures for disk access derived in section 10 to a B-tree with 20 character (average) unfactored keys and 1024 byte blocks gives an order 41 B-tree for the standard implementation and an order 52 B-tree for the DAC implementation. Using the same assumptions for average loading of the blocks used in the analysis, 1000 keys gives a 3 level standard B-tree and a 2 level

DAC B-tree. The expected cost is then 3 access for the standard B-tree and 4 for the DAC B-tree.

Increasing the size of the B-tree to 30,000 keys gives cost of 4 accesses for a 4 level standard B-tree and 54 for the 3 level DAC B-tree.

Obviously, the DAC B-tree will exhibit poorer performance, especially when the number of keys is large. This can be understood by recognizing that adding an additional level to a B-tree of order m results in a factor of m more keys with only one additional disk access for the standard implementation. For the DAC implementation, the access cost goes up linearly with the number of keys. The majority of this increased cost is mapping sites and reading tags. Additional work on the algorithms in this area (e.g, the SITE restart points described earlier) should result in sizeable gains.

8.4 More Performance Analysis is Possible

There are at least two interesting areas where more work could be done: 1) design and implement modifications to the algorithms to improve performance, and 2) experimental measurements of the performance could be made by instrumenting the implementation. Analytical measures of performance for update (insertion and deletion) could be derived. This might supply additional insight into the operation of the DAC algorithms, leading to better algorithms.

REFERENCES

1. Bayer, R., and McCreight, E. Organization and maintenance of large, ordered indexes. *Acta Informatica. 1* (1972), 173-189.

2. Brinch Hansen, P. *Operating System Principles.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

3. Comer, D. The ubiquitous b-tree. *ACM Comp. Surveys. 11*, 2 (June 1979), 121-137.

4. Cook, T. J. *An application of dynamic address computation in data management.* (Ph. D. Thesis) University of Utah, 1977.

5. Haerder, T. Implementation of a generalized access path structure for a relational database. *ACM Trans. on Database Sys. 3*, 3 (September 1978), 285-298.

6. Horowitz, E., and Sahni, S. *Fundamentals of data Structures.* Computer Science Press, Inc., Rockville, MD, 1976.

7. Knuth, D. E. *The Art of Computer Programming.* Addison-Wesley Publishing Co., Reading, MA, 1968.

8. Knuth, D. E. *The Art of computer Programming.* Addison-Wesley Publishing Co., Reading, MA, 1973.

9. Wiederhold, G. *Database Design.* McGraw-Hill, New York, NY, 1977.

10. West, R. *On the performance of b-trees using dynamic address computation.* (Masters Thesis) Virginia Polytechnic Institute and State University, 1985.