ON THE PERFORMANCE OF B-TREES USING DYNAMIC ADDRESS COMPUTATION

by

Raymond Troy West, Jr.

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

H. Rex Hartson, Chairman

Dennis Kafura

Robert Foutz

November, 1985

Blacksburg, Virginia
ON THE PERFORMANCE OF B-TREES USING DYNAMIC ADDRESS COMPUTATION

by

Raymond Troy West, Jr.

Committee Chairman: H. Rex Hartson
Computer Science

(ABSTRACT)

The B-tree is one of the more popular methods 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. An implementation of Dynamic Address Computation and a B-tree management package is described. Analytical performance measures are derived in an attempt to understand the performance characteristics of the B-tree. It is shown that the additional costs associated with Dynamic Address Computation result in an implementation that is competitive with traditional implementations only for small applications. For very large B-trees, additional work is required to make the performance acceptable. Some examples of possible modifications are discussed.
ACKNOWLEDGMENTS

I would like to thank a number of people who have encouraged me in this work. First, my wife who prodded and nudged me to continue when I was discouraged, along with my children who never complained when Dad left for long hours at the computer. Without their love and support, I would have quit long ago.

Second, my advisor, Dr. H. Rex. Hartson, who made many suggestions that improved the clarity and quality of the writing and who never gave up on me as the time stretched on and on.

Also, my Manager at Data General Corporation who let me use our computer lab for most of the work, both for the programming and the production of this thesis.

Last, but not least, my many friends at Calvary Evangelistic Center in Worcester, MA, who gave me their love and encouraged me to make this a priority, even over work in the church, and who offered many prayers on my behalf, both for the completion of this thesis and for me and my life as a child of God.
# TABLE OF CONTENTS

1. INTRODUCTION ........................................... 1
   1.1. The Role of B-trees in Database Management .... 2
   1.2. Thesis Organization .................................. 3

2. INTRODUCTION TO B-TREES ............................... 7

3. ANOTHER MODEL FOR B-TREES ............................ 13
   3.1. The Ragged Array .................................. 13
   3.2. The B-tree as a Ragged Array ...................... 16

4. B-TREE ALGORITHMS ...................................... 31
   4.1. Searching ......................................... 33
   4.2. Insertion .......................................... 38
   4.3. Deletion .......................................... 40
   4.4. Key Order Data Scan ................................ 43

5. SOME DATA MANAGEMENT TERMINOLOGY .................... 44

6. DYNAMIC ADDRESS COMPUTATION .......................... 48
   6.1. The Dynamic Address Computation Storage ........ 49
   6.2. Locating Data Stored in a DAC Structure .......... 61
   6.3. The Address of Data ................................ 69
   6.4. The Instance Equation ............................... 73
   6.5. Solving the Instance Equation ..................... 86
   6.5.1. Generalized Multiplication ...................... 86
   6.5.2. Generalized Division ............................ 88
   6.6. Storing the Data and Tags ......................... 89
   6.7. The SITE ........................................... 99
   6.8. Using the SITE and MAP_PAGE to Access Data .... 108
   6.9. The SCHEMA and DIRECTORY ......................... 121
   6.9.1. The SCHEMA .................................... 121
   6.9.2. The DIRECTORY .................................. 121
   6.10. Putting it all Together ............................ 125

7. IMPLEMENTING B-TREES USING DYNAMIC ADDRESS ....... 140
   7.1. The B-tree Definitions ............................. 140
   7.2. B-tree ............................................. 143
   7.3. N_KEYS ............................................ 146
   7.4. DATA ................................................ 148
   7.5. CHILD_NODE_INDEX .................................. 151
   7.6. INSERT_DATA and INSERT_KEY ....................... 153
   7.7. DELETE_DATA and DELETE_KEY ....................... 156
   7.8. NEW_ROOT .......................................... 159
   7.9. SPLIT_NODE ........................................ 159
   7.10. CONCAT_NODES ..................................... 162
   7.11. N_NODES .......................................... 162
# TABLE OF CONTENTS (Continued)

8. AN EXAMPLE OF A DAC B-TREE .......................... 165
   8.1. The Sample File Contents ........................... 172
   8.2. A Sample Address Computation ....................... 183
       8.2.1. Locate the Directory Data for the Directory 184
       8.2.2. Dynamic Address Computation-Tag Page. 185
       8.2.3. Dynamic Address Computation-Data Page 188
       8.2.4. Dynamic Address Computation-B-tree. 191
       8.2.5. Read the Data Using the SITE. 193
   8.3. Performance Summary. ............................... 194

9. PAGE MANAGEMENT AND PERFORMANCE IMPROVEMENTS ........... 199
   9.1. Page Management. .................................. 199
   9.2. Performance Improvement ......................... 200
       9.2.1. Pinning Blocks. .............................. 201
       9.2.2. Locally Factored Components ................ 202
       9.2.3. Restart Points. ............................... 205
       9.2.4. First Component Tag .......................... 207
       9.2.5. Restart Sites ................................. 208
       9.2.6. Splitting Pages ............................... 210

10. PERFORMANCE ANALYSIS ................................ 211
    10.1. Notation ....................................... 213
    10.2. Storage Requirements ............................ 214
    10.3. Runtime Performance ............................. 222
    10.4. An Example .................................... 236
        10.4.1. Storage Requirements Computation ........ 237
        10.4.2. Access Cost Computation ................. 241

11. CONCLUSIONS ........................................ 244
    11.1. It Worked ....................................... 244
    11.2. The B-tree Implementation was Easy .............. 245
    11.3. Hardware Considerations make a Difference .... 245
    11.4. The Performance Improvements make a Difference 246
    11.5. It Needs more Work ............................. 247
    11.6. More Performance Analysis is Possible .......... 249

12. REFERENCES ........................................ 250

Appendix A: GLOSSARY .................................. 251

VITA ................................................... 258
A B-tree is an m-way search tree which has been proposed and used for indexes and inverted files in database management systems [BAYER72] [COMED79] [HAERT78]. 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 [COOKT77] 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 introduces the concept of "pointer-free" B-trees with an algorithm to search a B-tree. He also discusses the implementation of the Dynamic Address Computation algorithms for retrieval of data. In this thesis, a complete implementation of the Dynamic Address Computation algorithms is presented and
used to build a complete "pointer-free" B-tree management package, to include 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.

1.1. The Role of B-trees in Database Management.

A database management system (DBMS) is a generalized system for storing and retrieving structured data. One of the important functions of a DBMS is to give access to data, based on the value of data items in the database. For example; "Retrieve the record with NAME = SMITH".

To avoid the cost of searching the entire database, various indexing methods have been devised which provide fast access to a record or data value. An indexing system is typically invoked by presenting it with an index value, called a key. The index will return either the record(s) which contain that key value in a designated field or some data that allows the record(s) to be found. The former system is called a primary index and the key is called a primary key. The latter is usually called a secondary index on a secondary key.
There are many methods of implementing an indexing system. Some candidates include:

1) Hashing
2) Linked Lists
3) Binary Search Trees
4) Sorted Lists
5) Indexed Sequential Access Method
6) B-trees

In database management, the current method of choice is B-trees and their variants. Some of the advantages claimed for B-trees are:

1) No reorganization of the index is ever required.
2) Data can be accessed in sorted order.
3) They are easily maintained.
4) Space is utilized efficiently.

1.2. Thesis Organization

This thesis presents a unique implementation of a B-tree indexing system, as introduced by Cook [COOKT77]. It is not concerned with the management of the database
records, as that is considered to be a separable problem. The system described performs the following functions:

1) Given a key value and some data value, place the key in a B-tree index and associate it with the data (the data is also stored and managed by the system). This data could be a record, a pointer to a record, a list of pointers to a set of records, or any other data that needs to be indexed.

2) Given a key value, retrieve the data associated with the key, using the B-tree index. A flag will be returned if the key is not in the index.

3) Given a key value, delete it and the associated data.

The remaining chapters of this thesis are organized as follows:

1) Chapter 2 is a general introduction to B-trees and their implementation.
2) Chapter 3 presents a model for B-trees that is an alternative to the traditional pointer-oriented model.

3) Chapter 4 describes the B-tree algorithms in terms of this model.

4) Chapter 5 introduces some general concepts of data management that will be useful in later discussions.

5) Chapter 6 is a detailed description of Dynamic Address Computation.

6) Chapter 7 describes how the Dynamic Address Computation procedures are used to implement the B-tree primitives required by the algorithms in chapter 4.

7) Chapter 8 contains an example of a complete B-tree index and its data, and uses the algorithms from chapter 7 to 'read' a key from the B-tree as an introduction to the problem of performance evaluation.
8) Chapter 9 discusses some modifications to the Dynamic Address Computation algorithms that can significantly improve its performance.

9) Chapter 10 describes a set of performance measures and derives an analytic model of the performance of the B-tree in terms of these measures.

10) Chapter 11 summarizes the results.
2. INTRODUCTION TO B-TREES

The definition of a B-tree uses other tree definitions which must be discussed first. All definitions and equations given here are similar to those found in [HOROE76].

Definition: A Tree is a finite set of one or more nodes such that:

1) There is a specially designated node called the root.

2) The remaining nodes are partitioned into \( n \leq 0 \) disjoint sets \( T_1, \ldots, T_n \), where each of the sets is a tree, called a subtree of the root.

The degree of a node is the number of subtrees of that node. The degree of a tree is the maximum degree of its nodes. The roots of the subtrees of a node are called its children and that node is called the parent of these nodes. A tree with no children is called an empty tree. Nodes with the same parent are called siblings. The descendants of a node are all of the nodes in its subtrees, and the ancestors of a node are all nodes on the path from
the root to that node. A leaf node is a node with no children.

The level of a node is recursively defined as:

1) The root of a tree is at level 1.

2) if a node is at level x all of its children are at level x+1.

The depth of a tree is the maximum level of its nodes.

Definition: An m-way search tree, T, is a tree in which all nodes are of degree m or less. If T is not empty, it has the following properties:

1) The root of T is a node which contains:

   \( n,(A_1,K_1), (A_2,K_2),..., (A_n,K_n), A(n+1) \)

   where the \( A_i, 1 \leq i \leq n+1 \) are the subtrees of T and the \( K_i, 1 \leq i \leq n \) are key values.

2) \( K_i < K(i+1), 1 \leq i < n. \)

3) All key values in subtree \( A_i \) are less than or equal to the key value \( K_i, 1 \leq i \leq n. \)
4) All key values in subtree $A(n+1)$ are greater than $K_n$.

5) The subtrees $A_i$, $1 \leq i \leq n+1$ are also $m$-way search trees.

In a DBMS the trees $T$ and $A_i$ are typically represented by pointers which are addresses of blocks on disk where the root node of the tree or subtree is found. The key values are field (attribute) values of the field which is being indexed. The root node of subtree $A_i$ is called the child node for key $K_i$. The last subtree ($A(n+1)$) is not associated with any key. The roots of the subtrees $A_i$, $1 \leq i \leq n+1$ are called the children of the node.

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

1) The root node has at least two children.

2) All nodes other than the root node and the leaf nodes have at least $[m/2]$ 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 [COMED79]). The algorithms here build a basic B-tree. The extension to B-tree variations is obvious, and is not precluded by 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, 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 call a B'-tree, and Comer calls a B+-tree, but, for simplicity, the generic term B-tree will continue to be used).
The contents of the data nodes are intentionally left unspecified. Later, it will be clear that a data node may contain a complete record, a primary key for a record, an address for a record or any other meaningful data that is to be indexed.

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 [HOROE76]:

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

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

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

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.
3. 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 fairly 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.

3.1. 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.
<table>
<thead>
<tr>
<th>Row</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A(1,1)</td>
<td>A(1,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A(2,1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A(3,1)</td>
<td>A(3,2)</td>
<td>A(3,3)</td>
<td>A(3,4)</td>
<td>A(3,5)</td>
<td>A(3,6)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A(4,1)</td>
<td>A(4,2)</td>
<td>A(4,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $A$:

$$A(1,1), A(1,2), A(2,1), A(3,1), A(3,2), A(3,3), A(3,4), \ldots, A(4,3).$$

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 (eg., $A(2,2)$) 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 [KNUTD75]. Again, pointers must be stored and links followed to find a given element.

This chapter will show a way that ragged arrays can be used to represent B-trees. The present discussion will not
consider implementation, as that is the subject of chapters 6 and 7.

3.2. 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:

\[ N_{\text{keys}}(i): \]
the number of keys in node \( i \).

\[ N_{\text{nodes}}: \]
the number of nodes in the B-tree.

\[ N_{\text{tree\_levels}}: \]
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 (implied in the above configuration) 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:

\[
\text{number of level } i \text{ nodes} = \sum_{k=1}^{1 + N_\text{keys} \cdot (\text{First}_\text{prev} + k - 1)}
\]

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, it 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 \( B_{\text{tree}}(i,j) \)) is at level \( y \) in the tree, the node index of the desired node (the \( j \)th 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:

\[
\begin{align*}
N_1 & \quad \text{for the root (1).} \\
+ N_2 & \quad \text{for the level 2 nodes} \\
+ N_3 & \quad \text{for the level 3 nodes} \\
+ \ldots & \\
+ N_y & \quad \text{for the level } y \text{ nodes} \\
+ N_{\text{keys}}(N_1+\ldots+N(y-1)+1)+1 & \quad \text{for the children of the first level } y \text{ node} \\
+ \ldots & \\
+ N_{\text{keys}}(i-1)+1 & \quad \text{for the children of the last level } y \text{ node before node } i \\
+ j & \quad \text{to get to the } j\text{-th child node of node } i.
\end{align*}
\]
But:

\[ N_1 = 1 \]

\[ \begin{align*}
N_2 &= \sum_{k=1}^{N_1} (N_{\text{keys}}(k) + 1) \\
N_3 &= \sum_{k=N_1+1}^{N_1+N_2} (N_{\text{keys}}(k) + 1) \\
N_4 &= \sum_{k=N_1+N_2+1}^{N_1+N_2+N_3} (N_{\text{keys}}(k) + 1)
\end{align*} \]

etc.

So \( N_1 + N_2 + N_3 + \ldots + N_y \) is actually:

\[
1 + \sum_{k=1}^{N_1+N_2+N_3+\ldots+N(y-1)} (N_{\text{keys}}(k) + 1)
\]

or, one plus the sum of \( N_{\text{keys}}(k)+1 \) 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 $B_{\text{tree}}(i,j)$ makes the total become:

$$\text{child\_node\_index} = 1 + j + \sum_{k=1}^{i-1} \left( 1 + N_{\text{keys}}(k) \right)$$

or

$$\text{child\_node\_index} = i + j + \sum_{k=1}^{i-1} \left( N_{\text{keys}}(k) \right) \quad (1)$$

Equation (1) gives the row index in the ragged array that represents the node associated with the key in $B_{\text{tree}}(i,j)$. 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, $B_{\text{tree}}(\text{child\_node\_index},1)$ through $B_{\text{tree}}(\text{child\_node\_index},N_{\text{keys}}(\text{child\_node\_index}))$ is the desired node.

To reference the last child node of node $i$, which contains keys greater than $B_{\text{tree}}(i,N_{\text{keys}}(i))$, $N_{\text{keys}}(i)+1$
is used for \( j \) in the above equation. The B-tree search algorithm uses equation (1) to traverse a B-tree stored in a ragged array.

As an example of this representation, and the use of equation (1), consider the three level B-tree shown in figure 3 (the order of this tree is four).
Figure 3: A B-tree
The single valued nodes at the lowest level represent the data nodes which contain the data indexed by the B-tree. They may not actually be a part of the B-tree but are included here for use in the examples. The desired result of a B-tree search is the index of the data node associated with the search key.

The ragged array representation for this B-tree is shown in figure 4.
<table>
<thead>
<tr>
<th>Node_index</th>
<th>N_keys</th>
<th>The Node</th>
<th>TREE LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10 20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>60 70 80</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1 5 8</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>12 15</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>21 22 23</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>51 55 57</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>64 66 68</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>71 75</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>85 90</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4. B-tree Stored as a Ragged Array
The data nodes are considered to be stored in a one dimensional array with one data node per array element. The first element is then the data for key 1, the second element the data for key 5, etc.

The B-tree modeled by this ragged array has the following characteristics:

1) The root node is node one.

2) The level two nodes are nodes two and three, and the level three nodes are nodes four through ten.

3) The number of level two nodes is $2 = 1 + N_{\text{keys}(1)}$.

4) The number of level three nodes is $7 = N_{\text{keys}(2)} + 1 + N_{\text{keys}(3)} + 1$. 
Equation (1) is applied in the following examples:

1) Compute the node index (array row index) for the child node associated with key 70, given that it is known from previous processing that this key is in $B_{\text{tree}}(3,2)$:

Equation (1) gives us:

\[
\text{child\_node\_index} = 3 + 2 + \sum_{k=1}^{2} N_{\text{keys}}(k)
\]

\[
= 3 + 2 + 1 + 2
\]

\[= 8\]

which is the desired result.

2) Compute the index of the data node associated with the last child node of node eight, i.e., the node whose key value is greater than all keys in node eight.
Thus, equation (1) becomes:

\[
\text{child\_node\_index} = 1 + 8 + N\_keys(8) + \sum_{k=1}^{7} N\_keys(k)
\]

\[
= 1 + 8 + 3 + 1 + 2 + 3 + 3 + 2 + 3 + 3
\]

\[
= 29
\]

This is the index of the node associated with the value "70". This is the correct index value if the data nodes are considered to be a part of the B-tree (i.e., the first data node is the 11th B-tree node). If they are not, the total number of nodes in the B-tree must be subtracted, resulting in the data index value of 19, which is correct. This example illustrates some of the flexibility of this model, as the indexed data can be stored either in the lowest level nodes of the B-tree, or in a separately indexed array.
4. B-TREE ALGORITHMS

There are four basic B-tree operations that will be considered. They are:

1) Searching a B-tree to find the index of the data associated with a given search key (and retrieving that data, if the key is found).

2) Inserting a new key and its associated data.

3) Deleting a key and its associated data.

4) Scanning all of the data in key order.

The algorithms are presented as pseudo PL/I procedures. Declarations of variables have been excluded in order to concentrate on the procedural aspects of the code.

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) \texttt{CHILD\_NODE\_INDEX(NODE\_INDEX,KEY\_INDEX)}:
returns the result of evaluating equation (1).

2) \texttt{INSERT\_DATA(DATA\_INDEX,NEW\_DATA)}:
inserts new data into the DATA array as a new record number DATA\_INDEX.

3) \texttt{INSERT\_KEY(NODE\_INDEX,KEY\_INDEX,NEW\_KEY)}:
inserts NEW\_KEY into the B-tree as B\_TREE(NODE\_INDEX,KEY\_INDEX).

4) \texttt{DELETE\_DATA(DATA\_INDEX)}:
deletes the element at DATA(DATA\_INDEX)

5) \texttt{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) \texttt{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 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.

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.

4.1. Searching

The search algorithm shown in figure 5 is a modification of the B-tree search algorithm given in [COOKT77]. 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, */
    KEY_INDEX = N_KEYS(START_NODE);
    KEY = B_TREE(START_NODE,KEY_INDEX);
    KEY_INDEX = KEY_INDEX; /* Last key */
    END;
ELSE DO; /* Start with the passed in parameters */
    KEY = B_TREE(START_NODE,START_KEY)
    KEY_INDEX = 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_INDEX /* 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 5: 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.

4.2. Insertion

The insertion algorithm (figure 6) 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 [HOROE76].
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 6: B-tree Insertion Algorithm
4.3. Deletion

The deletion algorithm (figure 7) is also an adaptation of an algorithm found in [HOROE76]. 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 */

Figure 7: B-tree Deletion Algorithm (Continued)
LEVEL = N_TREE_LEVELS;
DO WHILE (N_KEYS(I) < CEIL(M/2) - 1 AND I > 1);
   /* There are less than \( \lceil m/2 \rceil \) 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. */
               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 */
               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 = O;
      END;
      LEVEL = LEVEL - 1; /* Move up a level in the tree */
   END;
END;
4.4. 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.
5. SOME DATA MANAGEMENT TERMINOLOGY

The discussion of Dynamic Address Computation in Chapter 6 uses several familiar concepts in perhaps unfamiliar ways. The purpose of the present chapter is to describe some general aspects of data management in order to establish some terminology before it is used in the descriptions of Dynamic Address Computation.

In any data management system, there are four important entities that exist in one form or another. Depending on the generality of the system, they may be hard-wired into procedural code, or they may be implemented as tables and procedures that use these tables to control their execution.

First, there is a schema, which is a static description of the structure of the data being managed. In a program designed to manage only one fixed set of structures, the schema could be represented by the data declarations build into the program, for example, the COBOL DATA DIVISION.

In a generalized data management system, the schema might be one or more tables describing the format, maximum size, relative positions and other relationships between data items. These tables are then interpreted by a set of
procedures which manage the data. Many times (especially with variable length data), the schema values may only represent bounds. The actual values for some of these parameters (length, for example) may be different for different occurrences of the same data item type in the database. In this case, a second set of descriptive data is also required.

This data describes the actual lengths and counts of occurrences of data in a particular occurrence of a database described by a schema. In this thesis, each individual descriptive value is called a tag. For example, while the schema describes the format of records, there are tags indicating how many records there are of each type. While the schema specifies the maximum length of a varying length character string, there is a tag for each occurrence of that character string to describe its particular length. Again, depending on the generality of the data management system, some of the descriptive data may be incorporated directly into a procedure, some may be stored in system managed tables, and some stored with the data. For example, a program reading a file of fixed length sequential records may find the length of all records recorded once (eg., IBM's DCB), while the number of records may not be stored at all. Instead, the program just reads until an end-of-file marker is encountered.
An example of a common approach for variable length record files is to place the length of a record immediately before the record in the file (CDC SCOPE's W Control Word record format and the Data General AOS Variable Length record format). This format mixes the tags with the data. As will be seen in chapter 6, the tags may also be stored separately from the data they describe.

The third entity to be considered is a directory to map a data reference from a logical address (e.g., record number 500) to a physical address (e.g., block number 1975). This mapping may be very simple (block number = record number), or it may be very complex (e.g., CDC SCOPE's linked list of File Blocks or the multi-level index structure of Data General's AOS). The more complex directories will require that some extra data used to perform the mapping be stored, maintained and references to it mapped.

Of course, the data itself must be stored. By the time that the format and the contents of the schema, tags and directory have been defined, the format of the data is also mostly determined.

The data management system must accept a request for a piece of data and return the correct string of bits. It
uses the schema's description of the data and the directory to locate those bits on (for example) a particular track in a particular cylinder of a particular disk pack.
6. DYNAMIC ADDRESS COMPUTATION

Chapter 3 presented algorithms for manipulating a B-tree represented as a ragged array with a small amount of auxiliary information. It is easy to see that 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 [COOKT77], and it allows 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.

6.1. 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 8, taken from [COOKT77], 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 8: DAC Definition of the Structure W
This is the definition of a structure $W$. Figure 9 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 $Z$. 
Figure 9: Populated Occurrence of W
In this example, the sizes of each of the components are described using the syntactic construct "repeats (max P)" or "length (max Q)." 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 then, is actually a special case in which all components are factored.

In the a populated occurrence of W shown in figure 9, notice that there are a \( \leq A \) occurrences of X and Y and:

- \( Y(1) \) consists of \( c_1 \leq C \) occurrences of Z
- \( Y(2) \) consists of \( c_2 \leq C \) occurrences of Z
- \[ \cdots \]
- \( Y(a) \) consists of \( c_a \leq C \) occurrences of Z

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

The \( a, b_i, c_i, \) and \( d(i,j) \) are all \textit{tags} which represent the number of (sub)structure occurrences and the length of each of the data items. Figure 10 shows the collection of tags for the population shown in figure 9.
<table>
<thead>
<tr>
<th>Structure</th>
<th>Components</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 W</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>02 X</td>
<td>b1, b2, ... , ba</td>
<td></td>
</tr>
<tr>
<td>02 Y</td>
<td>c1, c2, ... , ca</td>
<td></td>
</tr>
<tr>
<td>03 Z</td>
<td>d(1,1), ... , d(1, c1), ... d(a, 1), ... , d(a, ca)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Tags for Instance of W
The information describing the structure \( W \) should require much less space than \( W \) 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 (directory or non-directory). 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 the directory and will be discussed in detail later in this chapter.

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 cl + 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 11 shows the order of the data for the populated structure.
Figure 11: Data Storage
6.2. 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, X(i) references the i-th X, Y(i) references the i-th "substructure" Y, and W(i).Z(j) 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.

There is more than one way that a reference to a particular string of bits may be expressed. The second X occurrence might be called W(2).X or W.X(2) or simply X(2). The second Y might be called W(2).Y or W.Y(2) or Y(2), and refers to all of the Z occurrences that make up the second Y. The third Z in the second Y can be called W(2).Y(3).Z or W(2).Y.Z(3) or W.Y(2).Z(3) or Y(2).Z(3) or W.Y.Z(2,3) or just Z(2,3). These are all just syntactic variations of W(2).Y(3).Z, which is the form that will be used here.

It is important to understand what a reference in this form really means. Referring to figure 9, several examples will be given to illustrate the important concepts.

Notice that W(3).X refers to a single leaf occurrence (the third X). Determining the location of W(3).X is a

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.

In general, a data structure may have more components than this simple example shows. In the manner of figure 8, a more general definition is shown in figure 12.
01 W
  .     repeats (max A),
  .     other substructures above X
  .
02 X
02 Y
  03 Z     length (max B),
  .     repeats (max C),
  .     length (max D),
  .     other substructures below Y
  .

Figure 12: A General Example of a DAC Structure
Here, a reference to \(W(I).Y\) would require that \(I\) occurrences of all descendants of \(W\) above (and including) \(X\) be considered, and only \(I-1\) occurrences of all substructures below (and including) \(Y\) be considered.

Notice that the substructure split occurs at the last component mentioned in the reference. Thus \(W(2).Y\) requires the split between \(W(2).X\) and \(W(2).Y\).

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.

To understand this better, consider the example from figure 11 and reference \(W(I).Y(J).Z\). Figure 13 illustrates how the various components contribute to the address of the desired datum (See figure 10 to review the tag values for
the structure). In this case, the substructure split occurs between components X and Y.
Figure 13: Address Computation for $W(I).Y(J).Z$
One more general example will be used to illustrate these concepts. Consider the structure defined in figure 14 and the reference C1(I1).C4(I4).C6. The structure is split at C6. This means that I1 complete occurrences of C2, C3 and C9 through C10 must be considered.
<table>
<thead>
<tr>
<th>C1</th>
<th>repeats (max R1),</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>length (max L2),</td>
</tr>
<tr>
<td>C3</td>
<td>length (max L3),</td>
</tr>
<tr>
<td>C4</td>
<td>repeats (max R4),</td>
</tr>
<tr>
<td>C5</td>
<td>length (max L5),</td>
</tr>
<tr>
<td>C6</td>
<td>length (max L6),</td>
</tr>
<tr>
<td>C7</td>
<td>repeats (max R7),</td>
</tr>
<tr>
<td>C8</td>
<td>length (max L8),</td>
</tr>
<tr>
<td>C9</td>
<td>repeats (max R9),</td>
</tr>
<tr>
<td>C10</td>
<td>length (max L10);</td>
</tr>
</tbody>
</table>

**Figure 14: A General Data Structure Example**
Then, Il-1 occurrences of C4 (since it is indexed) must be considered. Then I4 occurrences of C5 in the Il-st occurrence of C4 (in addition to the occurrences in the Il-1 previous occurrences) are added. Finally, I4-1 occurrences of C6, C7 and C8 are added.

6.3. The Address of Data

As the preceding discussions have implied, the address of a datum is the sum of the lengths of all data before it in the structure (in a pre-order sense). Referring to figure 11, the address of W(2).Y(3).Z is:

\[ b_1 + d(1,1) + \ldots + d(1,c_1) + b_2 + d(2,1) + d(2,2) \]

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:

\[
\sum_{i=1}^{I} b_i + \sum_{i=1}^{I-1} \left( \sum_{j=1}^{c_i} d(i,j) \right) + \sum_{j=1}^{J-1} d(I,j)
\]
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.

It is really just an optimization to add up the LEAF tags. In a more general sense, the leaf tags and the REP tags are the same. A REP tag represents the number of substructure occurrences that make up that REP component occurrence. A LEAF tag represents the number of BITS that make up that LEAF component occurrence. The only difference is that the "substructures" that make up LEAF components are always 1 bit long.

Using this analogy, the above equation becomes:

\[
\sum_{i=1}^{I} \left( \sum_{k=1}^{b_i} \right) + \sum_{i=1}^{I-1} \left( \sum_{j=1}^{c_i} \left( \sum_{k=1}^{d(i,j)} \right) \right) + \sum_{j=1}^{J-1} \left( \sum_{k=1}^{d(I,j)} \right)
\]
Thus, the length of any substructure occurrence is recursively defined as:

For a REP (sub)structure:

\[
\text{length of all immediate descendents} = \sum_{i=1}^{\text{tag}} \text{len. of 1st desc.} + \ldots + \sum_{i=1}^{\text{tag}} \text{len. of last desc.}
\]

For a LEAF substructure:

\[
\text{length of all immediate descendents} = \sum_{i=1}^{\text{tag}} \text{length of each bit} = \sum_{i=1}^{\text{tag}} 1 = \text{tag}
\]

The recursion terminates at the LEAF nodes.

Then, computing the address of a particular datum becomes a matter of evaluating this recursive equation by 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 DATASPN for that component. The DATASPN for a component is the sum of the tags for all occurrences of that component before the referenced datum. For a REP component, the DATASPN 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 DATASPN 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. As will be seen, the TAGSPAN for a component can be determined from the DATASPN of the parent (REP) component or from the DATASPN of the same component. The DATASPN for a component can be determined by adding up the first TAGSPAN tags for that component or from the TAGSPAN of a descendant component.

The relationship between the TAGSPAN and the DATASPN for a component is formally represented in the "Instance Equation".
6.4. 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 8, W is component 1 and Z is component 4). Now, the Instance Equation for component $k$ is:

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

The tags for the example structure (W) have been repeated in figure 15 for the following discussion.
<table>
<thead>
<tr>
<th>Structure</th>
<th>Components</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 W</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>02 X</td>
<td>b1, b2, ... , ba</td>
<td></td>
</tr>
<tr>
<td>02 Y</td>
<td>c1, c2, ... , ca</td>
<td></td>
</tr>
<tr>
<td>03 Z</td>
<td>d(1,1), ... , d(1, c1), ... , d(a,1), ... , d(a, ca)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 15: Tags for W (Repeated)
Now, assume that DATASPAN(3) (the DATASPAN for Y) has been computed. This DATASPAN value is the sum of some number of $c_i$. Since $c_i$ 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 $c_i$ tags that must be subtracted from DATASPAN(3) to reduce it to zero, or below.

The DAC process takes advantage of this close relationship between the DATASPAN value of a REP component and the TAGSPAN value of all of its immediate descendants to compute the DATASPAN value for all components, given a reference of the form previously discussed. Since the DATASPAN value for a LEAF component gives the total length of all occurrences of that component before the referenced datum, adding up the DATASPAN values for all LEAF components gives the offset to the datum.
The DAC process uses an "address table", contained in the schema and illustrated in figure 16 for the example structure. Two columns have been added, in addition to TAGSPAN and DATASPN. 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.
<table>
<thead>
<tr>
<th>TAGSPAN</th>
<th>DATASSPAN</th>
<th>INDEX</th>
<th>DONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 16: Example Address Table
The address table is initialized to reflect the form and content of the reference. A request such as \( W(2).Y(3).Z \) can be generalized to the form:

\[
\text{COMP}_a(I_a)\cdot\text{COMP}_b(I_b)\cdot\ldots\cdot\text{COMP}_z(I_z).
\]

Letters \( a - z \) are used instead of numbers \( 1 - n \) to emphasize that \( \text{COMP}_a \) is not necessarily component number 1, and all components in the structure may not be mentioned in the reference. Also, some components in the reference may not be given index values \( (I_x) \). The address table and two control values \( \text{TOP} \) and \( \text{BOTTOM} \) are then initialized according to the following rules:

1) \( \text{TOP} \) is set to the component number of \( \text{COMP}_a \).

2) \( \text{BOTTOM} \) is set to the component number of \( \text{COMP}_z \).

3) Only \( \text{COMP}_z \) may be of type \text{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 \( \text{COMP}_z \) is a \text{LEAF}, and has an index \( I_z \), \( I_z \) is assigned to the parent of \( \text{COMP}_z \).
5) DONE is set to "not done" for all components.

Using the example from figure 8, the reference $W(2).Z(3)$ results in the address table in figure 17. The reference to $W(2).Y(3).Z$ (the preferred notation) results in the same initial values.

The address computation process consists of filling in the empty spaces in the address table. The algorithm is shown in figure 18.
\begin{tabular}{|c|c|c|c|}
\hline
TAGSPAN & DATASPN & INDEX & DONE \\
\hline
W & & 2 & \sim \text{done} \\
\hline
X & & 0 & \sim \text{done} \\
\hline
Y & & 3 & \sim \text{done} \\
\hline
Z & & 0 & \sim \text{done} \\
\hline
\end{tabular}

TOP = 1 and BOTTOM = 4.

\textbf{Figure 17: 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 */ 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;
END;

Figure 18: The AC algorithm
The algorithm moves either "up" the structure, passing a TAGSPAN value up to be used as a DATASSPAN, or "down" the structure, passing a DATASSPAN value down to be used as a TAGSPAN. An input parameter (VALUE) represents the TAGSPAN or DATASSPAN 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 processing now depends on the direction.

1) Going up:

In this case, the input VALUE is the TAGSPAN of a descendant of component number COMP_NR. VALUE becomes the DATASSPAN for this component (DATASSPAN(COMP_NR)). The Instance Equation is solved for TAGSPAN(COMP_NR) (Solving the Instance Equation is discussed below). DONE(COMP_NR) is set to indicate that the component has been processed, going up, and the TAGSPAN value is passed up to the parent of component COMP_NR.

2) Going down:

Now the input VALUE is a DATASSPAN value being passed down from the parent of component number
COMP_NR. This DATASPAN value becomes the TAGSPAN value this component. If the (sub)structure is split at this component, because this component's parent was the last component mentioned in the reference, then VALUE - 1 is the new TAGSPAN value. If this (sub)structure occurrence is split, because there is a non-zero index value, VALUE - 1 is used, and INDEX(COMP_NR) is added to the result of solving the Instance Equation for DATASPAN(COMP_NR).

The algorithm is started by initializing the address table, as previously discussed, and the reference CALL AC(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 19.
<table>
<thead>
<tr>
<th>TAGSPAN</th>
<th>DATASPN</th>
<th>INDEX</th>
<th>DONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>b1+b2</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>2</td>
<td>c1+3</td>
<td>3</td>
</tr>
<tr>
<td>Z</td>
<td>c1+3</td>
<td>\ldots</td>
<td>0</td>
</tr>
</tbody>
</table>

TOP = 1 and BOTTOM = 4.

Figure 19: Completed Address Table
The distance to $Z(2,3)$ is the sum of the DATASPANs for the LEAF components, or:

$$b_1 + b_2 + d(1,1) + \ldots + d(2,2)$$

which is the result previously obtained.

6.5. 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.

6.5.1. Generalized Multiplication (GEN_MULT)

Solving for DATASPAN, given a TAGSPAN, involves simply summing TAGSPAN tag values. This is actually a generalization of the multiplication process, as can be
seen by considering the special case where \( \text{TAG}(k,i) = C \) for all \( i \). In this case:

\[
\text{DATASPN}(k) = \sum_{i=1}^{\text{TAGSPAN}(k)} \text{TAG}(k,i) = \sum_{i=1}^{\text{TAGSPAN}(k)} C = \text{TAGSPAN}(k) \times C
\]

Thus, solving for DATASPN at component \( k \) becomes either:

1) For factored components:

\[
\text{DATASPN}(k) = \text{TAGSPAN}(k) \times \text{TAG_VALUE}(k);
\]

2) For unfactored components:

\[
\text{DATASPN}(k) = \text{Sum of TAGSPAN}(k) \text{ tag values};
\]
6.5.2. Generalized Division (GEN_DIV)

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

\[
\text{DATASPAN}(k) = \text{TAGSPAN}(k) \times C
\]

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). The algorithm below results:

1) For factored components:

\[
\text{TAGSPAN}(k) = \text{DATASPAN}(k) / \text{TAG_VALUE}(k);
\]
\[
\text{REMAINDER}(k) = \text{DATASPAN}(k) \mod \text{TAG_VALUE}(k);
\]
\[
\text{IF } \text{REMAINDER}(k) > 0 \text{ THEN DO;}
\]
\[
\text{REMAINDER}(k) = \text{TAG_VALUE}(k) - \text{REMAINDER}(k);
\]
\[
\text{TAGSPAN}(k) = \text{TAGSPAN}(k) + 1;
\]
\[
\text{END;}
\]
2) For unfactored components:

```
TAGSPAN(k) = 0;
DO WHILE (DATASPAN(k) > 0);
    TAGSPAN(k) = TAGSPAN(k) + 1;
    DATASPAN(k) = DATASPAN(k) - TAG(TAGSPAN(k)));
END;
REMAINDER(k) = - DATASPAN(k);
```

6.6. Storing the Data and Tags

Both the tags and the data described by the tags must be stored. It could be stored in primary memory, but this thesis is directed at large databases which are stored on a direct access device such as disk or drum. 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 15, 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 20.

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 21.
Structure  Components  Tags

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01 W</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>02 X</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>02 Y</td>
<td>c1,c2, ... ,ca</td>
<td></td>
</tr>
<tr>
<td>03 Z</td>
<td>d(l,l), ... ,d(l,c1), ... d(a,l), ... ,d(a,ca)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 20: Tags for W with Factored X Component
Figure 21: 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 22. 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 22: 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 21 would be described by the page table in figure 23, where the actual values for \( a \), \( c_i \), \( w \), \( y \) and \( z \) must be inserted and the expressions evaluated, since only one number is stored in each position in the table.
Figure 23: 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 \((N - 1) \times z\). A form of generalized divide is now used to determine the correct page. The CLIQUE_LEN(i)'s for the third clique are added until the sum exceeds \((N - 1) \times z\). The number of CLIQUE_LEN 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 24 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 24: Data Page Table Structure
6.7. 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 25.
01 SITE,
  02 BLOCK ---
  02 PGE_OFFSET ---
  02 PGE_RESIDUE ---
  02 PGE_REV_RESIDUE ---
  02 PGE_INDEX ---
  02 MAP_INDEX ---;

Figure 25: 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 26, 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 26: 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 27, 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 27: Relationship Between PGE_INDEX and PGE_OFFSET

becomes (after Mapping)

(PGE_INDEX = 0)
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 shown in figure 28. To do the mapping, MAP_PAGE uses data from the tag page tables, which are part of the directory. As will be seen, this data is just another DAC structure, so to read it requires a SITE. This "directory site" will be assumed for now, and is called DIR_SITE. The directory site for access to the directory is supplied by the system, and is called the master site (MASTER_SITE).
MAP_PAGE: PROC(STRUC, SITE, DIR, DIR_SITE, CLIQUE);
/* Map an input SITE to a new SITE when the value of */
/* PGE_INDEX puts the data is a different page than */
/* MAP_INDEX. DIR = +1 for forward, -1 for backward. */

/* Compute the size of a page table entry. */
NEXT = LEN_NR + LEN_LEN + NR_CLIQUES * LEN_LEN;

/* Set up the Directory SITE (DIR_SITE) */
IF (DIR = FWD)
THEN DO; /* For forward mapping */
/* Point DIR_SITE at the next CLIQUE.LENG entry. */
SPGE_INDEX = MAP_INDEX * NEXT + LEN_NR + LEN_LEN +
(CLIQUE - 1) * LEN_LEN;
SUM_OF_LEN = PGE_RESIDUE; /* Move to end of page. */
END;
ELSE DO; /* For backward mapping */
/* Put DIR_SITE at end of previous CLIQUE.LENG entry */
SPGE_INDEX = (MAP_INDEX - 1) * NEXT -
(NR_CLIQUES - CLIQUE) * LEN_LEN;
/* Is the DIRECTORY data in the first page? */
IF SPGE_INDEX < SPGE_RESIDUE
THEN DO; /* YES - re-map here */
SPGE_OFFSET = SPGE_OFFSET + SPGE_INDEX;
SPGE_REV_RESIDUE = SPGE_REV_RESIDUE + SPGE_INDEX;
SPGE_RESIDUE = SPGE_RESIDUE - SPGE_INDEX;
SPGE_INDEX = 0;
END;
ELSE DO; /* NO - use MAP_PAGE for re-map of DIR_SITE */
CALL MAP_PAGE(DIRECTORY, DIR_SITE, FWD, MASTER_SITE, 1);
END;
/* Move to the beginning of the page. */
SUM_OF_LEN = PGE_REV_RESIDUE;
END;

Figure 28: Mapping a SITE (Continued)
/* Now set up the SITE */
DO WHILE (TRUE):
/* Move to next/prior page. */
MAP_INDEX = MAP_INDEX + DIR;

/* Get the length of the Clique in this page */
LAST_LEN =
GDATA(DIRECTORY,DIR_SITE,DIR,MASTER_SITE,LEN_LEN,1);

/* Calculate the total distance moved */
SUM_OF_LEN = SUM_OF_LEN + LAST_LEN;

IF SUM_OF_LEN > PGE_INDEX THEN DO; /* This is the page, so set up the SITE */
/* Find the page ID in the Page Table */
IF (DIR = FWD)
THEN SPGE_INDEX = LEN_LEN + (CLIQUE-1)*LEN_LEN;
ELSE SPGE_INDEX = LEN_LEN * CLIQUE + LEN_LEN;
/* Read the ID */
BLOCK_NR =
GDATA(DIRECTORY,DIR_SITE,BCK,MASTER_SITE,LEN_NR,1);
IF (DIR = FWD)
THEN DO; /* Reset the SITE for Forward Mapping */
PGE_REV_RESIDUE = PGE_INDEX-(SUM_OF_LEN - LAST_LEN);
PGE_RESIDUE = SUM_OF_LEN - PGE_INDEX;
END;
ELSE DO; /* Reset the SITE for Backward Mapping */
PGE_REV_RESIDUE = SUM_OF_LEN - PGE_INDEX;
PGE_RESIDUE = PGE_INDEX - (SUM_OF_LEN - LAST_LEN);
END;
PGE_INDEX = 0;
PGE_OFFSET = PGE_REV_RESIDUE;
SPGE_INDEX = LEN_LEN;

/* Compute the PGE_OFFSET */
DO I = 1 TO CLIQUE - 1;
CL_LENG =
GDATA(DIRECTORY,DIR_SITE,FWD,MASTER_SITE,LEN_LEN,1);
PGE_OFFSET = PGE_OFFSET + CL_LENG;
SPGE_INDEX = LEN_LEN; /* Next Length */
END;
RETURN;
END;
ELSE DO; /* Not done, so move to the next page */
SPGE_INDEX = NEXT;
END;
END; /* DO WHILE (TRUE); */
END MAP_PAGE

Figure 28:  Mapping a SITE (Concluded)
The previous discussion implied that the SITE is used by knowing where it is currently mapped and setting PGE_INDEX to the distance to the desired data. This is true in many cases and is used heavily in MAP_PAGE. It must also be possible to find a datum knowing only its distance from the beginning of the clique. This is accomplished by setting all of the SITE to zero except for PGE_INDEX, which is set to the distance to the data. MAP_PAGE will then correctly map the site.

6.8. Using the SITE and MAP_PAGE to Access Data

Once a SITE has been determined, it can be used to reference the data (again, reference means read, insert, delete or replace). There are four corresponding algorithms to accomplish this.

In the first part of each algorithm, the SITE is mapped (using MAP_PAGE if necessary) so that PGE_OFFSET marks the beginning of the data. Each algorithm performs its assigned task, and then the site is checked to insure that it is returned in a consistent state so that the calling routines can use the relative addressing described in the discussion of the SITE. This state requires that the PGE_OFFSET mark the data with PGE_INDEX = zero.
We will discuss the algorithm to read data (GDATA) in some detail (see figure 29). The other algorithms have much in common with GDATA, so their discussions will concentrate on only the differences.
Figure 29: Data Retrieval Algorithm

GDATA:PROC(STRUC,SITE,DIR,DIR_SITE,LENG,CLIQUE);
/* Get the LENG bits in STRUC addressed by SITE. */

IF ((DIR = FWD & PGE_INDEX > PGE_RESIDUE) |
    (DIR = BCK & PGE_INDEX + LENG > PGE_REV_RESIDUE))
    THEN DO; /* Must re-map to a new page */
        IF (DIR = BCK) THEN PGE_INDEX = PGE_INDEX + LENG;
        CALL MAP_PAGE(STRUC,SITE,DIR,DIR_SITE,CLIQUE);
    END;
ELSE DO; /* Remap can be done within the Page. */
    IF (DIR = FWD)
        THEN DO;
            PGE_OFFSET = PGE_OFFSET + PGE_INDEX;
            PGE_RESIDUE = PGE_RESIDUE - PGE_INDEX;
            PGE_REV_RESIDUE = PGE_REV_RESIDUE + PGE_INDEX;
            END;
    ELSE DO; /* Map it so it looks like FWD */
            PGE_OFFSET = PGE_OFFSET - (PGE_INDEX + LENG);
            PGE_RESIDUE = PGE_RESIDUE + (PGE_INDEX + LENG);
            PGE_REV_RESIDUE = PGE_REV_RESIDUE - (PGE_INDEX + LENG);
        END;
    PGE_INDEX = 0;
END;
/* Have a SITE — Now go get the data. */
IF (LENG < PGE_RESIDUE) /* If all data in this page */
    THEN RETURN(READ(BLOCK_NR,PGE_OFFSET,LENG)) /* Get it*/
ELSE DO;
    /* Set up a new SITE. The data is not all in one page. */
    NEW SITE = SITE;
    NPGE_INDEX = PGE_RESIDUE; /* Read will start at end */
    RETURN (READ(BLOCK_NR,PGE_OFFSET,PGE_RESIDUE) | |
          GDATA(STRUC,NEW SITE,FWD,DIR_SITE,
          LENG-PGE_RESIDUE,
          CLIQUE)); /* Recursively get the rest */
END;
/* Have the data — Now may need to reset the SITE */
IF (DIR = BCK) /* If this SITE was adjusted to FWD */
    THEN DO; /* Reset it to be a backward read */
        IF (LENG < PGE_RESIDUE) /* If end is on this page */
            THEN DO; /* The reset is easy */
                PGE_OFFSET = PGE_OFFSET + LENG;
                PGE_RESIDUE = PGE_RESIDUE - LENG;
                PGE_REV_RESIDUE = PGE_REV_RESIDUE + LENG;
            END;
        ELSE DO; /* Remap is needed to move to the next page */
            PGE_INDEX = LENG;
            CALL MAP_PAGE(STRUC,SITE,FWD,DIR_SITE,CLIQUE);
        END;
    END;
END;
A request to read data is accomplished by first setting the site (either by setting SITE to all zeroes except for PGE_INDEX or by setting PGE_INDEX to the relative distance to the desired data). Then, the request is read as:

1) When DIR = FWD, get the LENG bits in clique CLIQUE starting PGE_INDEX bits beyond the current site location.

2) When DIR = BCK, get the LENG bits in clique CLIQUE ending PGE_INDEX bits before the current site location.

The first thing that the algorithm does is remap the site so that PGE_OFFSET marks the data and PGE_INDEX is zero. To simplify the actual reading of the data, the site is always set to mark the beginning of the data, even for a backward read.

The check for whether or not the data begins in the current page is different for forward and backward reads. For a forward read, the data is in the same page if PGE_INDEX is less than PGE_RESIDUE. For a backward read, PGE_INDEX refers to the end of the data, with LENG being the length, so PGE_INDEX + LENG must be less than or equal
to PGE_REV_RESIDUE. Notice that for a site set to zeros a remapping is always required, even for a reference to the first bit, when PGE_INDEX is also zero.

If remapping is required, MAP_PAGE is used. For a backward reference, PGE_INDEX is increased by LENG to map to the beginning of the data.

If the data begins in the current page, PGE_OFFSET, PGE_RESIDUE and PGE_REV_RESIDUE are all modified by PGE_INDEX. PGE_INDEX is always set to zero.

After the above operations, PGE_OFFSET is the real distance from the beginning of the page to the beginning of the data. The block number is BLOCK_NR. If all LENG bits are in this page (LENG \leq PGE_RESIDUE), the data is simply read from disk. If some of the data is in subsequent pages, a new site is set up with PGE_INDEX set to PGE_RESIDUE (the distance to the beginning of the remainder of the data). The PGE_RESIDUE bits in the current page are read. GDATA is then used recursively to retrieve the rest of the data, which is concatenated to the end of the data found in this page.

After the data has been retrieved, if the original request was for a backward read, the site is reset to point
to the end of the data read (this is expected by the calling program).

Inserting data (figure 30) is similar, except that the site is the address of the point where the LENG bits are to be inserted.
IDATA:PROC(STRUC,SITE,DIR,DIR_SITE,LENG,CLIQUE,DATA)

RECURSIVE;

/* Insert LENG bits into STRUC at SITE */
IF ((MAP_INDEX = 0) |
(DIR = FWD & PGE_INDEX > PGE_RESIDUE) |
(DIR = BCK & PGE_INDEX > PGE_REV_RESIDUE))
THEN CALL MAP_PAGE(STRUC,SITE,DIR,DIR_SITE,CLIQUE);
ELSE DO; /* Remap can be done within the SITE */
IF (DIR = FWD)
THEN DO;
PGE_OFFSET = PGE_OFFSET + PGE_INDEX;
PGE_RESIDUE = PGE_RESIDUE - PGE_INDEX;
PGE_REV_RESIDUE = PGE_REV_RESIDUE + PGE_INDEX;
END;
ELSE DO; /* Map it so it looks like FWD */
PGE_OFFSET = PGE_OFFSET - PGE_INDEX;
PGE_RESIDUE = PGE_RESIDUE + PGE_INDEX;
PGE_REV_RESIDUE = PGE_REV_RESIDUE - PGE_INDEX;
END;
PGE_INDEX = 0;
END;

/* Have a SITE - Now go insert the data */
AVAIL_SPACE = PAGE_SIZE - PGE_LENGTH(MAP_INDEX);
IF (LENG g AVAIL_SPACE) /* If it fits in this page */
THEN DO; /* Insert it */
CALL INSERT(BLOCK_NR,PGE_OFFSET,LENG,DATA);
PGE_RESIDUE = PGE_RESIDUE + LENG; /* Adjust the SITE */
CALL MOD_PGE_LENG(MAP_INDEX,CLIQUE,DIR_SITE,LENG);
END;
ELSE DO; /* Harder Insert - Find more space */
CALL SPLIT_PGE(SITE,CLIQUE,DIR_SITE);
CALL INSERT(BLOCK_NR,PGE_OFFSET,LENG,DATA);
PGE_RESIDUE = PGE_RESIDUE + LENG;
CALL MOD_PGE_LENG(MAP_INDEX,CLIQUE,DIR_SITE,LENG);
END;

IF (DIR = BCK) /* If SITE was adjusted to FWD */
THEN DO; /* Reset it to look like a backward insert */
IF (LENG < PGE_RESIDUE) /* If end is on this page */
THEN DO; /* Then the reset is easy */
PGE_OFFSET = PGE_OFFSET + LENG;
PGE_RESIDUE = PGE_RESIDUE - LENG;
PGE_REV_RESIDUE = PGE_REV_RESIDUE + LENG;
END;
ELSE DO; /* Remap is needed to move to next page */
PGE_INDEX = LENG;
CALL MAP_PAGE(STRUC,SITE,FWD,DIR_SITE,CLIQUE);
END;
END;
END;

Figure 30: Data Insertion Algorithm
In order to insure that forward insertion at the end of a page does not result in an extra remapping to the beginning of the next page (which does not exist if this is the end of the structure), the insert point is considered to be in the same page when PGE_INDEX is less than or equal to PGE_RESIDUE. Also, LENG is not considered in the backward insert since PGE_INDEX marks the point where the data ends and the current length of the data in the structure is zero (it has not been inserted yet). Since the insertion of the first bit into an empty structure occurs with PGE_INDEX = PGE_RESIDUE = 0, a check for MAP_INDEX equal zero is included to trigger a remapping in this case.

If the data being inserted will fit on the block (BLOCK_SIZE - PGE_LENGTH(MAP_INDEX) > LENG), the insertion is simple. If there is not room on the block, the page is split into two pages, each one-half of a block long. This requires that a new page table entry be inserted into the directory for this structure. This is accomplished in SPLIT_PGE by recursively using IDATA with DIR_SITE as the site.

Conversly, deleting data (figure 31) may result in an empty block. This block is returned to the free block pool and the MAP_INDEXth entry in the page table of this structure is deleted, using DDATA. Notice that the SITE
does not need to be reset after the deletion since the "beginning" and "end" of the data are the same point (the data's length is zero after the deletion).
Figure 31: Data Deletion Algorithm
The first step in replacing an existing entry (Figure 32) is to determine if the old and new entry are the same length. If they are not, DDATA and IDATA are used to delete the old entry and insert the new one. If the data has not changed length, the old data is simply overwritten. This less costly form of replacement is very common, especially in the updating of tags, which are fixed length within a clique.
RDATA:

PROC(STRUC,SITE,DIR,DIR_SITE,LENG,OLD_LENG,CLIQUE,DATA)
RECURRENCE;

/* Replace the OLD_LENG bits in STRUC at SITE with the */
/* LENG bits of new data */
/* If the data has changed length then delete and insert */
IF LENG ≠ OLD_LENG
THEN DO;
    CALL DDATA(STRUC,SITE,DIR,DIR_SITE,OLD_LENG,CLIQUE);
    CALL IDATA(STRUC,SITE,DIR,DIR_SITE,LENG,CLIQUE,DATA);
END;
ELSE DO; /* Otherwise, just overwrite the old data. */
    IF ((DIR = FWD & PGE_INDEX > PGE_RESIDUE) |
        (DIR = BCK & PGE_INDEX+LENG > PGE_REV_RESIDUE))
    THEN DO;
        IF (DIR = BCK) THEN PGE_INDEX = PGE_INDEX + LENG;
        CALL MAP_PAGE(STRUC,SITE,DIR,DIR_SITE,CLIQUE);
    END;
    ELSE DO; /* Remap can be done within the SITE */
        IF (DIR = FWD)
        THEN DO;
            PGE_OFFSET = PGE_OFFSET + PGE_INDEX;
            PGE_RESIDUE = PGE_RESIDUE - PGE_INDEX;
            PGE_REV_RESIDUE = PGE_REV_RESIDUE + PGE_INDEX;
        END;
        ELSE DO; /* Map it so it looks like FWD */
            PGE_OFFSET = PGE_OFFSET - (PGE_INDEX + LENG);
            PGE_RESIDUE = PGE_RESIDUE + (PGE_INDEX + LENG);
            PGE_REV_RESIDUE = PGE_REV_RESIDUE-(PGE_INDEX+LENG);
        END;
    PGE_INDEX = 0;
END;

Figure 32: Data Replacement Algorithm (Continued)
/* Have a SITE - Now go replace the data */
IF (LENG < PGE_RESIDUE) /* If data is in one page */
/* Then simply replace it */
THEN CALL WRITE(BLOCK_NR, PGE_OFFSET, LENG, DATA);
ELSE DO;
/* The data is not in one page. Set up a new SITE. */
NEW_SITE = SITE;
NPGE_INDEX = PGE_RESIDUE; /* Repl will start at end */
NEW_LENG = LENG - PGE_RESIDUE;
/* Replace the first PGE_RESIDUE bits of data. */
CALL WRITE (BLOCK_NR, PGE_OFFSET, PGE_RESIDUE,
SUBSTR(DATA, 1, PGE_RESIDUE));
/* Recursively replace the rest */
CALL RDATA (STRUC, NEW_SITE, FWD, DIR_SITE, NEW_LENG,
NEW_LENG, CLIQUE, SUBSTR(DATA, PGE_RESIDUE + 1, NEW_LENG));
END;
IF (DIR = BCK) /* Is SITE was adjusted to FWD */
THEN DO; /* Reset to to look like a backward replace. */
IF (LENG < PGE_RESIDUE) /* If end is on this page */
THEN DO; /* Then the reset is easy */
PGE_OFFSET = PGE_OFFSET + LENG;
PGE_RESIDUE = PGE_RESIDUE - LENG;
PGE_REV_RESIDUE = PGE_REV_RESIDUE + LENG;
END;
ELSE DO; /* Remap is needed to move to the next page */
PGE_INDEX = LENG;
CALL MAP_PAGE (STRUC, SITE, FWD, DIR_SITE, CLIQUE);
END;
END;
END;

Figure 32: Data Replacement Algorithm (Concluded)
6.9. The SCHEMA and DIRECTORY

Several times in the preceding discussion, reference has been made to the schema and directory structures, and pieces have been described. They will now be presented in their entirety.

6.9.1. The SCHEMA

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 33 (the actual data types are not specified, as they are implementation dependent).

6.9.2. 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 34.
Figure 33: The SCHEMA Structure
Figure 34: 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.
6.10. 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.

The general form of this algorithm is shown in figure 35. 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.
IF this is a request for user data
    THEN locate the directory data for this structure.
    ELSE return the known location of the directory data for the directory.
Locate the data, using AC.

Figure 35: Locating the Directory Data
To satisfy a request for a data location first 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 DATASSPAN values for all 
LEAF components. The detailed algorithm is shown in figure 
36.
AC_LOC:PROC(STRUC,INPUTS,SITE,DIR_C_SITE,DIR_D_SITE);
/*
  IF TYPE_STRUCT != DIREC_STRUCT /* 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 + DATASPN(LEV);
END;
END;

Figure 36: Locating the Data
Once the data is located, the appropriate routine (GDATA, IDATA, DDATA or RDATA) can be used to reference the data. The tags associated with the structure must also be updated. The details of this maintenance will not be discussed here. The interested reader is referred to [COOKT77].

The algorithms to reference data are given in figures 37 through 40, without additional comment. The only functions used that have not been discussed here are some tag access routines used for tag maintenance (ex. FAST_TAG). These routines take advantage of the fact that the tag page tables in the directory have already been located. Otherwise, they are identical to the regular tag access algorithms.
AC_READ:PROC(STRUC,INPUTS);
/* Read the DAC structure STRUC using INPUTS */

/* Locate the data */
CALL AC_LOC(STRUC,INPUTS,SITE,DIR_C_SITE,DIR_D_SITE);

/* Get its length */
LENG = FAST_TAG(STRUC,TAGSPAN(BOTTOM)+1,DIR_C_SITE);

/* Get the data */
RETURN (GDATA(STRUC,SITE,FWD,DIR_D_SITE,LENG,1));
END;

Figure 37: Reading from a DAC Structure
AC_INS: PROC(STRUC, INPUTS, NEW_TAGS, ICOMP, DATA, LENG);
/* Insert DATA into STRUC using INPUTS */

/* Locate the insert point */
CALL AC_LOC(STRUC, INPUTS, SITE, DIR_C_SITE, DIR_D_SITE);

/* Increment the tag at the insert component */
TAG = FAST_TAG(STRUC, ICOMP, TAGSPAN(ICOMP),
    DIR_C_SITE) + 1;
CALL FAST_REPT(STRUC, ICOMP, TAGSPAN(ICOMP),
    DIR_C_SITE, TAG);

/* Insert the new tags at the lower components */
DO I = 1 TO N_NEW_TAGS;
    CMP = ICOMP + I;
    /* Only insert tags into UNFACTORED cliques */
    IF (FORMAT(LEV) = UNF)
        THEN DO;
        CALL FAST_INST(STRUC, LEV, TAGSPAN(LEV) + TAG_NR(I),
            DIR_C_SITE, N_TAG(I));
        END;
    END;

/* Now insert the data */
CALL IDATA(STRUC, SITE, FWD, DIR_D_SITE, LENG, 1, DATA);
END;

Figure 38: Inserting into a DAC Structure
AC_DEL:PROC(STRUC,INPUTS,DCOMP);
    /* Delete data from STRUC using INPUTS */

    /* Locate the beginning of the data */
    CALL AC_LOC(STRUC,INPUTS,SITE,DIR_C_SITE,DIR_D_SITE);
    LOC1 = PGE_INDEX;
    /* Save the address cut */
    DO CMP = 1 TO NR_COMPS;
        BEG_ADDRESS.TAGSPAN(LEV) = ADDRESS_TABLE.TAGSPAN(LEV);
    END;

    /* Index to the next substructure occurrence */
    INDEX(DCOMP) = INDEX(DCOMP) + 1;
    /* Locate the end of the data */
    CALL AC_LOC(STRUC,INPUTS,SITE,DIR_C_SITE,DIR_D_SITE);
    /* Address of the end of the data */
    LOC2 = PGE_INDEX;

    /* Decrement the tag at the deletion component */
    TAG = FAST_TAG(STRUC,DCOMP,TAGSPAN(DCOMP),
                  DIR_C_SITE) - 1;
    CALL FAST_REPT(STRUC,DCOMP,TAGSPAN(DCOMP),
                    DIR_C_SITE,TAG);

    /* Delete all tags associated with the deleted data */
    DO CMP = DCOMP + 1 TO EXTENT(DCOMP);
        IF (FORMAT(LEV) = UNF)
            THEN DO;
                /* First tag */
                N1 = BEG_ADDRESS.TAGSPAN(LEV) + 1;
                /* Last tag */
                N2 = ADDRESS_TABLE.TAGSPAN(LEV);
                DO TAG_NR = N1 TO N2;
                    CALL FAST_DELT(STRUC,LEV,TAG_NR,DIR_C_SITE);
                END;
            END;
        END;

    /* Restore the SITE to the beginning of the data */
    /* Notice that the rest of the SITE is still 0 */
    PGE_INDEX = LOC1;

    /* Delete the data */
    CALL DDATA(STRUC,SITE,FWD,DIR_D_SITE,LOC2-LOC1,1);
END;

Figure 39: Delete Data from a DAC Structure
AC_REPL:PROC(STRUC,INPUTS,DATA,LENG):
  /* Replace data in STRUC using INPUTS */

  /* Locate the data */
  CALL AC_LOC(STRUC,INPUTS,SITE,DIR_C_SITE,DIR_D_SITE);

  /* Get the current length */
  OLD_LENG = FAST_TAG(STRUC,BOTTOM,TAGSPAN(BOTTOM)+1,
                       DIR_C_SITE);

  /* Reset the tag if the length has changed */
  IF OLD_LENG ≠ LENG
    THEN DO;
      CALL FAST_REPT(STRUC,BOTTOM,TAGSPAN(BOTTOM)+1,
                      DIR_C_SITE,LENG);
    END;

  /* Replace the data */
  CALL RDATA(STRUC,SITE,FWD,DIR_D_SITE,LENG,OLD_LENG,1,DATA);
END;

Figure 40: Replacing Data in a DAC Structure
Tags must be referenced to get data lengths, the number of occurrences of a sub-structure, etc. To reference the Nth tag, AC_LOC is used to find the page tables (DIR_C_SITE) for the tags and a site set to zeros with PGE_INDEX set to \((N - 1) \times \text{TAG}\_\text{SIZE}\). Then either GDATA, IDATA, DDATA or RDATA is used to perform the proper operation (See figures 41 through 44).

The algorithms in figures 37 through 44 will be used in the next chapter to build a set of routines that model a B-tree as a ragged array.
AC_TAG:PROC(STRUC,TCOMP,TINDEX) RETURNS (FIXED BIN);
/* Read the TINDEX'th tag for STRUC at component TCOMP */

/* Get the size of the tag */
LENG = TAG_SIZE(TCOMP);

/* If a FACTORED tag, the tag is in the DIRECTORY */
IF FORMAT(TCOMP) = FACT
THEN DO;
    TAG = TAG_VALUE(TCOMP) * TAG_UNITS(TCOMP);
    END;
ELSE DO; /* Must get tag from the clique */
    /* Locate the DIRECTORY data */
    CALL AC_LOC(DIRECTORY,INPUTS,DIR_C_SITE,MAST_C_SITE,
                MAST_D_SITE);
    CALL MAP_PAGE(DIRECTORY,DIR_C_SITE,FWD,MAST_D_SITE,1);

    /* Set up a SITE */
    BLOCK_NR = 0;
    PGE_OFFSET = 0;
    PGE_RESIDUE = 0;
    PGE_REV_RESIDUE = 0;
    MAP_INDEX = 0;
    PGE_INDEX = (TINDEX - 1) * LENG;

    /* Get the tag */
    TAG = GDATA(STRUC,SITE,FWD,DIR_C_SITE,LENG,
                CLIQUE_ID(TCOMP));
    TAG = TAG * TAG_UNITS(TCOMP);
    END;
RETURN(TAG);
END;

Figure 41: Reading a Tag
AC_INST:PROC(STRUC,TCOMP,TINDEX,TAG);
/* Insert a new TINDEX'th tag at component TCOMP in STRUC */
  /* Get the length of the tag */
  LENG = TAG_SIZE(TCOMP);

  /* Locate the DIRECTORY data */
  CALL AC_LOC(DIRECTORY,INPUTS,DIR_C_SITE,MAST_C_SITE,
              MAST_D_SITE);
  CALL MAP_PAGE(DIRECTORY,DIR_C_SITE,FWD,MAST_D_SITE,1);

  /* Set up a SITE */
  BLOCK_NR = 0;
  PGE_OFFSET = 0;
  PGE_RESIDUE = 0;
  PGE_REV_RESIDUE = 0;
  MAP_INDEX = 0;
  PGE_INDEX = (TINDEX - 1) * LENG;

  /* Insert the tag */
  TTAG = TAG / TAG_UNITS(TCOMP);
  CALL IDATA(STRUC,SITE,FWD,DIR_C_SITE,LENG,
             CLIQUE_ID(TCOMP),TTAG);
END;

Figure 42: Inserting a Tag
AC_DELT:PROC(STRUC,TCOMP,TINDEX);
/* Delete the TINDEX'th tag at component TCOMP for STRUC */

/* Get the length of the tag */
LENG = TAG_SIZE(TCOMP);

/* Locate the DIRECTORY data */
CALL AC_LOC(DIRECTORY,INPUTS,DIR_C_SITE,MAST_C_SITE,
           MAST_D_SITE);
CALL MAP_PAGE(DIRECTORY,DIR_C_SITE,FWD,MAST_D_SITE,1);

/* Set up a SITE */
BLOCK_NR = 0;
PGE_OFFSET = 0;
PGE_RESIDUE = 0;
PGE_REV_RESIDUE = 0;
MAP_INDEX = 0;
PGE_INDEX = (TINDEX - 1) * LENG;

/* Delete the TAG */
CALL DDATA(STRUC,SITE,FWD,DIR_C_SITE,LENG,CLIQUE_ID(TCOMP));
END;

Figure 43: Deleting a Tag
AC_REPT:PROC(STRUC,TCOMP,TINDEX,TAG);
/* Replace the TINDEX'th tag at component TCOMP in STRUC */

/* Get the length of the tag */
LENG = TAG_SIZE(TCOMP);

/* Locate the DIRECTORY data */
CALL AC_LOC(DIRECTORY,INPUTS,DIR_C_SITE,MAST_C_SITE,
            MAST_D_SITE);
CALL MAP_PAGE(DIRECTORY,DIR_C_SITE,FWD,MAST_D_SITE,1);

/* Set up a SITE */
BLOCK_NR = 0;
PGE_OFFSET = 0;
PGE_RESIDUE = 0;
PGE_REV_RESIDUE = 0;
MAP_INDEX = 0;
PGE_INDEX = (TINDEX - 1) * LENG;

/* Replace the tag */
TTAG = TAG / TAG_UNITS(TCOMP);
CALL RDATA(STRUC,SITE,FWD,DIR_C_SITE,LENG,LENG,
            CLIQUE_ID(TCOMP),TTAG);
END;

Figure 44: Replacing a Tag
7. 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. In this chapter, these procedures are specified in some detail. 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.

7.1. The B-tree Definitions

Using the notation in the previous chapters, the array B_TREE for a B-tree of degree m with varying length keys is defined in figure 45.

Similarly, figure 46 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 45: DAC Definition of the B-TREE Array
01 DATA_NODE repeats (max MAXDATA),
02 DATA length (max DATA_LENGTH):

Figure 46: DAC Definition of the DATA Array
Now, to reference keys or data requires only that the correct AC procedure be executed to find the data. The following algorithms use dynamic address computation procedures and data to perform their operation. Again, many details of implementation, such as declarations, have been omitted for clarity.

7.2. B-tree

The B-tree "array" is now replaced by a pair of procedures. One is a function, B_TREE (figure 47), which returns the specified key value. The other is REPLACE_KEY (figure 48), which corresponds to the assignment B_TREE(NODE_INDEX, KEY_INDEX) = NEW_KEY.
B_TREE: PROC(NODE_INDEX, KEY_INDEX);
/* Set INPUTS to access NODE(NODE_INDEX).KEY(KEY_INDEX) */
RETURN(AC_READ(B_TREE, INPUTS));
END; /* END B_TREE */

Figure 47: B_TREE Function
REPLACE_KEY: PROC(NODE_INDEX, KEY_INDEX, B_TREE_KEY);

/* Set INPUTS to access NODE(NODE_INDEX).KEY(KEY_INDEX) */
CALL AC_REPL(B_TREE, INPUTS, B_TREE_KEY, LENGTH(B_TREE_KEY));
END; /* END REPLACE_KEY */

Figure 48: REPLACE_KEY Procedure
7.3. N_KEYS

The array N_KEYS need not be explicitly stored, since the dynamic addressing data includes the number of keys in each node as the tags at the KEYS-PER-NODE component. So, N_KEYS can be a simple function that returns the tag value (figure 49). Notice that N_KEYS is never explicitly changed by SEARCH_B_TREE, INSERT_B_TREE or DELETE_B_TREE. Since the value of N_KEYS is a DAC Tag, and the DAC algorithms maintain the tags, these changes occur automatically when keys are added or deleted.
N_KEYS: PROC (NODE_INDEX);

    RETURN (AC_TAG (B_TREE, KPN_COMP, NODE_INDEX));

END;

Figure 49: N_KEYS Function
7.4. DATA

The array DATA is replaced by two procedures similar to the ones that replaced B_TREE. They are given in figures 50 and 51.
DATA:PROC(DATA_INDEX);

/* Set INPUTS to access DATA(DATA_INDEX) */
RETURN(AC_READ(DATA, INPUTS));
END; /* END DATA */

Figure 50: DATA Function
REPLACE_DATA: PROC(DATA_INDEX, NEW_DATA);

/* Set INPUTS to access DATA(DATA_INDEX) */
CALL AC_REPL(DATA, INPUTS, NEW_DATA, LENGTH(NEW_DATA));
END; /* END REPLACE_DATA */

Figure 51: REPLACE_DATA Procedure
7.5. CHILD_NODE_INDEX

A careful look at the algorithm SEARCH_B_TREE, which uses this procedure, will show that CHILD_NODE_INDEX is called only after a key has been retrieved from the current node using the procedure B_TREE, defined above. It is also easy to see that the DATASSPAN value for the KEYS_PER_NODE component (calculated by AC) will be:

\[
\text{NODE_INDEX} - 1 + \sum_{k=1}^{\text{NODE_INDEX} - 1 + \sum_{k=1}^{\text{KEY_INDEX}} \text{N.Keys} (k)}
\]

The value required is the result of evaluating equation (1) in chapter 3. But, for B_TREE(NODE_INDEX,KEY_INDEX), equation (1) would be:

\[
\text{NODE_INDEX} + \text{KEY_INDEX} + \sum_{k=1}^{\text{NODE_INDEX} - 1 + \sum_{k=1}^{\text{KEY_INDEX}} \text{N.Keys} (k)}
\]

So, the simple function in figure 52 results.
CHILD_NODE_INDEX: PROC(NODE_INDEX, KEY_INDEX);
    RETURN(NODE_INDEX + B_TREE.DATASPN(KPN_COMP));
END;

Figure 52: CHILD_NODE_INDEX Function
7.6. INSERT_DATA and INSERT_KEY

The procedures to insert data and keys both use the corresponding dynamic address computation procedures to insert into a structure. These algorithms are shown in figures 53 and 54.
INSERT_DATA:PROC(DATA_INDEX,NEW_DATA);

    /* Set INPUTS to access DATA(DATA_INDEX) */
    /* Set NEW_TAGS as the length of the data */
    CALL AC_INS(DATA,INPUTS,NEW_TAGS,DATA_NODE_COMP,
                NEW_DATA,LENGTH(DATA));

END; /* END INSERT_DATA */

Figure 53: INSERT_DATA Procedure
INSERT_KEY: PROC(NODE_INDEX, KEY_INDEX, B_TREE_KEY);

/* Set INPUTS to access NODE(NODE_INDEX).KEY(KEY_INDEX) */
/* Set NEW_TAGS as the length of the new key */

CALL AC_INS(B_TREE, INPUTS, NEW_TAGS, KPN_COMP,
            B_TREE_KEY, LENGTH(B_TREE_KEY));

END; /* END INSERT_KEY */

Figure 54: INSERT_KEY Procedure
7.7. DELETE_DATA and DELETE_KEY

The procedures to delete data and keys are also simple applications of more general dynamic address computation procedures. Their algorithms are given in figures 55 and 56. The DELETE_KEY procedure also checks for the deletion of the last key in the root node. If this occurs, the node is deleted and N_TREE_LEVELS decremented by one. Notice that the root is the only node that will ever have its last key deleted (Assuming m>4).
DELETE_DATA: PROC (DATA_INDEX);

/* Set INPUTS to access DATA (DATA_INDEX) */
CALL AC_DEL (DATA, INPUTS, DATA_NODE_COMP);
END; /* END DELETE_DATA */

Figure 55: DELETE_DATA Procedure
DELETE_KEY: PROC(NODE_INDEX, KEY_INDEX);

/* Set INPUTS to access NODE(NODE_INDEX).KEY(KEY_INDEX) */

IF N_KEYS(NODE_INDEX) = 1 & NODE_INDEX = 1 THEN DO; /* Deleting the last key from the root node */
    DEL_COMP = NODE_COMP; /* Delete the entire node */
    N_TREE_LEVELS = N_TREE_LEVELS - 1; /* Shrink the tree */
END;
ELSE DO; /* Not deleting the last key */
    DEL_COMP = KPN_COMP; /* Delete only the key */
END;

CALL AC_DEL(B_TREE, INPUTS, DEL_COMP);

END; /* END DELETE_KEY */

Figure 56: DELETE_KEY Procedure
7.8. NEW_ROOT

A B-tree grows at its root. When a new level is added by the insert procedure, it is done by inserting a new root node with one key and incrementing N_tree_levels. The procedure to do this is shown in figure 57.

7.9 SPLIT_NODE

Splitting a node requires only that a new tag for the KEYS_PER_NODE component be created immediately after the tag for the node to be split. The N_KEYS value for the node is divided between the nodes, but no data is moved. The key which is to be deleted is deleted first. Figure 58 shows this procedure.
NEW_ROOT: PROC(NEW_KEY);
/* Create a new node with NEW_KEY as its only key */
/* Set NEW_TAGS to be 1 at the KEYS_PER_NODE component, */
/* and the length of the key an the KEY component */
/* Set INPUTS to access NODE(1) */

CALL AC_INS(B_TREE, INPUTS, NEW_TAGS, NODE_COMP, NEW_KEY);

N_TREE_LEVELS = N_TREE_LEVELS + 1;

END;

Figure 57: NEW_ROOT Procedure
SPLIT_NODE: PROC(NODE_INDEX, KEY_INDEX);

/* Get the number of keys */
N_KEYS = AC_TAG(B_TREE, KPN_COMP, NODE_INDEX);

/* Delete the key */
CALL DELETE_KEY(NODE_INDEX, KEY_INDEX);

/* Replace the old tag with the (new) number of keys to */
/* be in node number NODE_INDEX   (KEY_INDEX - 1)   */
CALL AC_REPT(B_TREE, KPN_COMP, NODE_INDEX, KEY_INDEX - 1);

/* Insert a new tag for the new node with remaining keys*/
CALL AC_INST(B_TREE, KPN_COMP, NODE_INDEX+1, N_KEYS-KEY_INDEX);

/* Increment the number of nodes */
N_KEYS = AC_TAG(B_TREE, NODE_COMP, 1) + 1;
CALL AC_REPT(B_TREE, NODE_COMP, 1, N_KEYS);

END;

Figure 58: SPLIT_NODE Procedure
7.10. CONCAT_NODES

This procedure concatenates two adjacent nodes into one. Again, no data is move. Only tags are changed. The new key is added to the first node before the nodes are concatenated (figure 59).

7.11. N_NODES

The number of nodes in a tree is the tag at the NODE component of the B-tree structure. It is automatically maintained by the AC_procedures. The B-tree procedures only need to read its value, so the function N_NODES (figure 60) returns the current value.
CONCAT_NODES: PROC(NODE_INDEX, NEW_KEY);
/* Concatenate nodes NODE_INDEX and NODE_INDEX + 1, */
/* adding NEW_KEY between the original keys in the nodes */

/* Get the key index for the new key in the node */
N1 = AC_TAG(B_TREE, KPN_COMP, NODE_INDEX) + 1;

/* Insert the key */
CALL INSERT_KEY(NODE_INDEX, N1, NEW_KEY);

/* Number of keys in the second node */
N2 = AC_TAG(B_TREE, KPN_COMP, NODE_INDEX+1);

/* Assign all of the keys to the first node */
CALL AC_REPT(B_TREE, KPN_COMP, NODE_INDEX, N1+N2);

/* Delete only the tag for the second node */
CALL AC_DELT(B_TREE, KPN_COMP, NODE_INDEX+1);

/* Decrement the number of nodes */
N1 = AC_TAG(B_TREE, NODE_COMP, 1) - 1;
CALL AC_REPT(B_TREE, NODE_COMP, 1, N1);
END;

Figure 59: CONCAT_NODES Procedure
N_NODES:PROC;
/* Return the tag for the NODE component. This is the */
/* number of nodes. */

RETURN (AC_TAG(B_TREE,NODE_COMP,1));
END;

Figure 60: N_NODES Function
8. AN EXAMPLE OF A DAC B-TREE

The entire DAC concept, when considered as a whole, is quite complex. It is often difficult to see how individual pieces fit together, even when the pieces are clear. In order to demonstrate a complete address computation, this chapter has an example of a DAC file holding a B-tree and associated data. Working out the entire process from start to finish exercises most of the DAC concepts. In addition, a concrete example serves to demonstrate the issues involved in performance evaluation of the system.

To make the example relevant to the implementation being described here, the B-tree from the example in chapter 3 is stored into a DAC file (the B-tree is repeated in figure 61). To make the example more interesting, the page (block) size is artificially small, resulting in more pages than would be the case in the real system. Also, most of the blocks are not as full as would normally be expected, so the number of page table entries is large enough to make the DIRECTORY non-trivial.
Figure 61: A Sample B-tree
The data nodes are arbitrarily assigned data derived directly from the key. For example, the data indexed by the key "5" is the string "FIVE". This creates fairly long data (relative to the keys) that is of varying length. The example uses varying length representations (unfactored) for both the keys and the data, as this results in the most interesting results.

For ease of reference, the definitions of the Directory, Schema, B-tree and Data structures are repeated in figures 62 thru 65. They are all useful in interpreting the discussion that follows.
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_LENGTH 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 62: The DIRECTORY Structure
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 Components */
  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 no., 0 for first */
    03 CLIQUE_ID ----, /* CLIQUE no. for UNF Comp only */
  02 ADDRESS_TABLE (NR_COMPONENTS), /* One per component*/
    03 TAGSPAN ----,
    03 DATASPN ----,
    03 INDEX ----,
    03 DONE ----;

Figure 63: The SCHEMA Structure
01 NODE repeats (max MAXNODES),
02 KEYS_PER_NODE repeats (max m-1),
03 KEY length (max KEY_LENGTH);

Figure 64: DAC Definition for the B-tree
01 DATA_NODE repeats (max MAX_DATA),
  02 DATA length (max DATA_LENGTH);

Figure 65: DAC Definition for the Data
8.1. The Sample File Contents

The entire file is given first, as it is nearly impossible to develop it step by step. The interactions between the contents of various pieces of the file is recursive (or at least iterative) since the DIRECTORY structure describes itself in its first set of substructure occurrences.

The example uses the following parameters:

1) Every page table number is 13 bits long (the number used in the real system).

2) Every Block is 320 bits long. (The real system used 4096).

This results in the tags for the B_TREE, DATA and DIRECTORY data structures shown in figure 66. The DIRECTORY tags are the result of the distribution of data into pages described later.

Notice that the tags that give the length of the KEYS and DATA are all a number that represents the number of characters, not bits. This is because this data is always a
multiple of 8 bits long, so the DESCRIPTORS for these components contain TAG_UNITS values of 8. The tags are multiplied by TAG_UNITS before being used (See the AC_TAG algorithm in chapter 6).

Then, the data and tags are rather arbitrarily distributed among several pages to produce some non-trivial page tables. The DIRECTORY's page tables and tags are in the first three pages, and the rest of the page tables, tags and data are given in figure 67.
<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>TAGS (Tag stored in DESCRIPTORS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 DIRECTORY</td>
<td></td>
</tr>
<tr>
<td>02 TAG_PAGES</td>
<td>3</td>
</tr>
<tr>
<td>03 BLOCK_NR</td>
<td>1</td>
</tr>
<tr>
<td>03 PGE_LENGTH</td>
<td>3</td>
</tr>
<tr>
<td>03 CLIQUES</td>
<td>(13)</td>
</tr>
<tr>
<td>04 CLIQUE_LENG</td>
<td>3</td>
</tr>
<tr>
<td>02 DATA_PAGES</td>
<td>8</td>
</tr>
<tr>
<td>03 BLOCK_NR</td>
<td>4</td>
</tr>
<tr>
<td>03 PGE_LENGTH</td>
<td>3</td>
</tr>
<tr>
<td>03 CLIQUE_LENG</td>
<td>(13)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>01 NODE</td>
<td>10</td>
</tr>
<tr>
<td>02 KEYS_PER_NODE</td>
<td>2</td>
</tr>
<tr>
<td>03 KEY</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>01 DATA_NODE</td>
<td>24</td>
</tr>
<tr>
<td>02 DATA</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 66: Tags for the Example B-tree
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>3</td>
<td>BLOCK_NR(1,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>110</td>
<td>PGE_LENGTH(1,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>10</td>
<td>CLIQUE_LENGTH(1,1,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>36</td>
<td>CLIQUE_LENGTH(1,1,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>28</td>
<td>CLIQUE_LENGTH(1,1,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>36</td>
<td>CLIQUE_LENGTH(1,1,4)</td>
</tr>
</tbody>
</table>

(Page Table for the Directory Tags)

| 2    | 13  | 1     | BLOCK_NR(1,1) |
|      | 13  | 78    | PGE_LENGTH(1,1) |
|      | 13  | 78    | CLIQUE_LENGTH(1,1) |
|      | 13  | 2     | BLOCK_NR(1,2) |
|      | 13  | 312   | PGE_LENGTH(1,2) |
|      | 13  | 312   | CLIQUE_LENGTH(1,2) |
|      | 13  | 4     | BLOCK_NR(1,3) |
|      | 13  | 195   | PGE_LENGTH(1,3) |
|      | 13  | 195   | CLIQUE_LENGTH(1,3) |
|      | 13  | 5     | BLOCK_NR(1,4) |
|      | 13  | 156   | PGE_LENGTH(1,4) |
|      | 13  | 156   | CLIQUE_LENGTH(1,4) |
|      | 13  | 6     | BLOCK_NR(1,5) |
|      | 13  | 156   | PGE_LENGTH(1,5) |
|      | 13  | 156   | CLIQUE_LENGTH(1,5) |
|      | 13  | 7     | BLOCK_NR(1,6) |
|      | 13  | 156   | PGE_LENGTH(1,6) |
|      | 13  | 156   | CLIQUE_LENGTH(1,6) |
|      | 13  | 13    | BLOCK_NR(1,7) |
|      | 13  | 195   | PGE_LENGTH(1,7) |
|      | 13  | 195   | CLIQUE_LENGTH(1,7) |
|      | 13  | 16    | BLOCK_NR(1,8) |
|      | 13  | 195   | PGE_LENGTH(1,8) |
|      | 13  | 195   | CLIQUE_LENGTH(1,8) |

Figure 67: Contents of the Example B-tree (Continued)
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 (DIRECTORY Tags)</td>
<td>10</td>
<td>3</td>
<td>Tag - DIRECTORY Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1</td>
<td>Tags - TAG_PAGES Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>Tags - CLIQUES Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>Tags - CLIQUES Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>Tags - CLIQUES Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2</td>
<td>Tags - CLIQUES Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2</td>
<td>Tags - CLIQUES Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2</td>
<td>Tags - CLIQUES Comp.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>8</td>
<td>Tags - DATA_PAGES Comp.</td>
</tr>
<tr>
<td>4 (B-tree Tag Page Table)</td>
<td>13</td>
<td>9</td>
<td>BLOCK_NR(2,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>108</td>
<td>PGE_LENGTH(2,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>10</td>
<td>CLIQUE_LENG(2,1,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>50</td>
<td>CLIQUE_LENG(2,1,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>48</td>
<td>CLIQUE_LENG(2,1,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>11</td>
<td>BLOCK_NR(2,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>114</td>
<td>PGE_LENGTH(2,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>CLIQUE_LENG(2,2,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>50</td>
<td>CLIQUE_LENG(2,2,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>64</td>
<td>CLIQUE_LENG(2,2,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>12</td>
<td>BLOCK_NR(2,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>80</td>
<td>PGE_LENGTH(2,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>CLIQUE_LENG(2,3,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>CLIQUE_LENG(2,3,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>80</td>
<td>CLIQUE_LENG(2,3,3)</td>
</tr>
</tbody>
</table>

Figure 67: Contents of the Example B-tree (Continued)
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>13</td>
<td>17</td>
<td>BLOCK_NR(2,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>88</td>
<td>PGE_LENGTH(2,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>88</td>
<td>CLIQUE LENG(2,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>19</td>
<td>BLOCK_NR(2,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>128</td>
<td>PGE_LENGTH(2,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>128</td>
<td>CLIQUE LENG(2,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>21</td>
<td>BLOCK_NR(2,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>88</td>
<td>PGE_LENGTH(2,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>88</td>
<td>CLIQUE LENG(2,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>23</td>
<td>BLOCK_NR(2,4)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>56</td>
<td>PGE_LENGTH(2,4)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>56</td>
<td>CLIQUE LENG(2,4)</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>8</td>
<td>BLOCK_NR(3,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>90</td>
<td>PGE_LENGTH(3,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>10</td>
<td>CLIQUE LENG(3,1,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>80</td>
<td>CLIQUE LENG(3,1,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>10</td>
<td>BLOCK_NR(3,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>56</td>
<td>PGE_LENGTH(3,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>CLIQUE LENG(3,2,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>56</td>
<td>CLIQUE LENG(3,2,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>BLOCK_NR(3,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>56</td>
<td>PGE_LENGTH(3,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0</td>
<td>CLIQUE LENG(3,3,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>56</td>
<td>CLIQUE LENG(3,3,2)</td>
</tr>
</tbody>
</table>

Figure 67: Contents of the Example B-tree (Continued)
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>13</td>
<td>15</td>
<td>BLOCK_NR(3,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>PGE_LENGTH(3,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>CLIQUE_LENG(3,1)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>18</td>
<td>BLOCK_NR(3,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>PGE_LENGTH(3,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>CLIQUE_LENG(3,2)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>20</td>
<td>BLOCK_NR(3,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>96</td>
<td>PGE_LENGTH(3,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>96</td>
<td>CLIQUE_LENG(3,3)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>22</td>
<td>BLOCK_NR(3,4)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>PGE_LENGTH(3,4)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>CLIQUE_LENG(3,4)</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>24</td>
<td>DATA_NODE Component Tag</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3</td>
<td>DATA Component Tag - 1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>DATA Component Tag - 2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5</td>
<td>DATA Component Tag - 3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3</td>
<td>DATA Component Tag - 4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6</td>
<td>DATA Component Tag - 5</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>DATA Component Tag - 6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6</td>
<td>DATA Component Tag - 7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>DATA Component Tag - 8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>DATA Component Tag - 9</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12</td>
<td>DATA Component Tag - 10</td>
</tr>
</tbody>
</table>

Figure 67: Contents of the Example B-tree (Continued)
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>10</td>
<td>NODE Component Tag</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1</td>
<td>K_P_N Component Tag - 1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
<td>K_P_N Component Tag - 2</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>K_P_N Component Tag - 3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>K_P_N Component Tag - 4</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
<td>K_P_N Component Tag - 5</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 5</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 6</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>5</td>
<td>DATA Component Tag - 11</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>DATA Component Tag - 12</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>DATA Component Tag - 13</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>11</td>
<td>DATA Component Tag - 14</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6</td>
<td>DATA Component Tag - 15</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>DATA Component Tag - 16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>DATA Component Tag - 17</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>3</td>
<td>K_P_N Component Tag - 6</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>K_P_N Component Tag - 7</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>K_P_N Component Tag - 8</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
<td>K_P_N Component Tag - 9</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
<td>K_P_N Component Tag - 10</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>KEY Component Tag - 7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>KEY Component Tag - 8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>KEY Component Tag - 9</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 10</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 11</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 12</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 13</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 14</td>
</tr>
</tbody>
</table>

Figure 67: Contents of the Example B-tree (Continued)
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 15</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 17</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 18</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 19</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 20</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 21</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 22</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 23</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>KEY Component Tag - 24</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>24</td>
<td>BLOCK_NR(3,5)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>64</td>
<td>PGE_LENGTH(3,5)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>64</td>
<td>CLIQUE_LENG(3,5)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>25</td>
<td>BLOCK_NR(3,6)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>120</td>
<td>PGE_LENGTH(3,6)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>120</td>
<td>CLIQUE_LENG(3,6)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>26</td>
<td>BLOCK_NR(3,7)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>128</td>
<td>PGE_LENGTH(3,7)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>128</td>
<td>CLIQUE_LENG(3,7)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>27</td>
<td>BLOCK_NR(3,8)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>88</td>
<td>PGE_LENGTH(3,8)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>88</td>
<td>CLIQUE_LENG(3,8)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>28</td>
<td>BLOCK_NR(3,9)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>120</td>
<td>PGE_LENGTH(3,9)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>120</td>
<td>CLIQUE_LENG(3,9)</td>
</tr>
</tbody>
</table>

Figure 67: Contents of the Example B-tree (Continued)
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 (DATA Tags - Third Page)</td>
<td>8</td>
<td>10</td>
<td>DATA Component Tag - 18</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7</td>
<td>DATA Component Tag - 19</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>DATA Component Tag - 20</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>DATA Component Tag - 21</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6</td>
<td>DATA Component Tag - 22</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>11</td>
<td>DATA Component Tag - 23</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6</td>
<td>DATA Component Tag - 24</td>
</tr>
</tbody>
</table>

15 (DATA Data page - 1) 104 ONEFIVEEIGHTT

<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 (DATA Data Page Table - Third Page)</td>
<td>13</td>
<td>29</td>
<td>BLOCK_NR(3,10)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>PGE_LENGTH(3,10)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>104</td>
<td>CLIQUE_LENG(3,10)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>30</td>
<td>BLOCK_NR(3,11)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>96</td>
<td>PGE_LENGTH(3,11)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>96</td>
<td>CLIQUE_LENG(3,11)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>31</td>
<td>BLOCK_NR(3,12)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>112</td>
<td>PGE_LENGTH(3,12)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>112</td>
<td>CLIQUE_LENG(3,12)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>32</td>
<td>BLOCK_NR(3,13)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>160</td>
<td>PGE_LENGTH(3,13)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>160</td>
<td>CLIQUE_LENG(3,13)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>33</td>
<td>BLOCK_NR(3,14)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>120</td>
<td>PGE_LENGTH(3,14)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>120</td>
<td>CLIQUE_LENG(3,14)</td>
</tr>
</tbody>
</table>

17 (B-tree Data page-1) 88 50102060708

18 (DATA Data page-2) 104 ENTWELVEFIFTE

Figure 67: Contents of the Example B-tree (Continued)
<table>
<thead>
<tr>
<th>PAGE</th>
<th>LEN</th>
<th>VALUE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 (B-tree Data page-2)</td>
<td>128</td>
<td>0158121521222351</td>
<td></td>
</tr>
<tr>
<td>20 (DATA Data page-3)</td>
<td>96</td>
<td>ENTWENTYTWNEN</td>
<td></td>
</tr>
<tr>
<td>21 (B-tree Data page-3)</td>
<td>88</td>
<td>55576466687</td>
<td></td>
</tr>
<tr>
<td>22 (DATA Data page-4)</td>
<td>64</td>
<td>TY ONETWENTY</td>
<td></td>
</tr>
<tr>
<td>23 (B-tree Data page-4)</td>
<td>56</td>
<td>1758590</td>
<td></td>
</tr>
<tr>
<td>24 (DATA Data page-5)</td>
<td>64</td>
<td>TWOTWENT</td>
<td></td>
</tr>
<tr>
<td>25 (DATA Data page-6)</td>
<td>120</td>
<td>Y THREEFIFTYFIF</td>
<td></td>
</tr>
<tr>
<td>26 (DATA Data page-7)</td>
<td>128</td>
<td>TY ONEFIFTY FIVE</td>
<td></td>
</tr>
<tr>
<td>27 (DATA Data page-8)</td>
<td>88</td>
<td>FIFTY SEVEN</td>
<td></td>
</tr>
<tr>
<td>28 (DATA Data page-9)</td>
<td>120</td>
<td>SIXTYSIXTY FOUR</td>
<td></td>
</tr>
<tr>
<td>29 (DATA Data page-10)</td>
<td>104</td>
<td>SIXTY SIXSIXT</td>
<td></td>
</tr>
<tr>
<td>30 (DATA Data page-11)</td>
<td>96</td>
<td>Y EIGHTSEVEN</td>
<td></td>
</tr>
<tr>
<td>31 (DATA Data page-12)</td>
<td>112</td>
<td>TYSEVENTY ONES</td>
<td></td>
</tr>
<tr>
<td>32 (DATA Data page-13)</td>
<td>160</td>
<td>EVENTY FIVEEIGHTY EI</td>
<td></td>
</tr>
<tr>
<td>33 (DATA Data page-14)</td>
<td>120</td>
<td>GHTY FIVENINETY</td>
<td></td>
</tr>
</tbody>
</table>

Figure 67: Contents of the Example B-tree (Concluded)
8.2. A Sample Address Computation

The sample address computation reads the second key in the fourth node of the B-tree (that is, NODE(4).KEYS_PER_NODE(2).KEY). The value to be read is "5".

To review, the steps required are:

1) Locate the directory data (page tables) for the B-tree data structure (this is setting the DIRECTORY Sites):

   a) Locate the directory data (page tables) for the DIRECTORY structure (the MASTER Sites).

   b) Perform Dynamic Address Computation in the Directory to locate the tag page tables for the B-tree structure.

   c) Perform Dynamic Address Computation in the Directory to locate the data page tables for the B-tree structure.
2) Perform Dynamic Address Computation in the B-tree structure to locate the desired datum (Setting the SITE).

3) Read the data using the SITE.

The discussion is organized according to the above outline, and the cost of performing each step recorded. At the end, a complete read of the data will have been performed. The recorded costs give an approximation of the total cost.

8.2.1. Locate the Directory Data for the DIRECTORY

The directory is a self describing structure, and the page tables that describe it are the first occurrence of the tables in the DIRECTORY. These tables are always at the same location and are located by the MASTER Sites. These sites (one for the tag page tables and one for the data page tables) are always maintained in memory, and are immediately available at essentially no cost. What actually happens is that the DIRECTORY Sites for the address computation about to happen are just set to the values in the MASTER Sites.
8.2.2. Dynamic Address Computation - Tag Page Tables

DAC requires that a page table be initialized, and the AC algorithm be executed on that table using the DIRECTORY tags. Locating the beginning of the tag page tables for the B-tree is equivalent to locating DIRECTORY(2).TAG_PAGES(1), since the B-tree is the second structure described by the DIRECTORY. The address table is initialized to all zeroes, except for the INDEX values for DIRECTORY and TAG_PAGES, which are set to 2 and 1, respectively. The final address table, after executing AC is shown in figure 68.
<table>
<thead>
<tr>
<th>TAGSPAN</th>
<th>DATASPIN</th>
<th>INDEX</th>
<th>DONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECTORY</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>TAG_PAGES</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BLOCK_NR</td>
<td>1</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>PGE_LENGTH</td>
<td>1</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>CLIQUES</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>CLIQUE_LENGTH</td>
<td>4</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>DATA_PAGES</td>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>BLOCK_NR</td>
<td>8</td>
<td>104</td>
<td>0</td>
</tr>
<tr>
<td>PGE_LENGTH</td>
<td>8</td>
<td>104</td>
<td>0</td>
</tr>
<tr>
<td>CLIQUE_LENGTH</td>
<td>8</td>
<td>104</td>
<td>0</td>
</tr>
</tbody>
</table>

TOP = 1    BOTTOM = 2

Figure 68: Address Table for Tag Page Table
Most of the components of this structure are Factored, and the tags are stored in the DESCRIPTORS. This means that no disk accesses are required to retrieve these tags. The only tags that must be read are:

1 Tag for the DIRECTORY Component (in block 3).
1 Tag for the TAG_PAGES Component (in block 3).
1 Tag for the CLIQUES Component (in block 3).
1 Tag for the DATA_PAGES Component (in block 3).

Reading the tags at a level consists of initializing a SITE.INDEX to the offset to the tag in the CLIQUE \(((\text{Tag\_number}-1)\times\text{Tag\ size})\) and then mapping that site to the page containing the tag. This requires 3, 4, 5 and 6 references to the DIRECTORY Tag Page Table in block 1.

Adding up the DATASPAWN values for the LEAF Components gives the total displacement in the DIRECTORY Data Structure to the beginning of the Tag Page Tables for the NODE (B-tree) Structure:

\[13+13+52+104+104+104 = 390 \text{ bits.}\]

Mapping this site to the page table location requires 5 references to the directory Data Page Table (in block 2).
This represents 3 reads of PGE_LEN to determine which page contains the desired page table, one read of CLIQUE_LEN to get the size of the page containing the table, and one read of BLOCK_NR to get the number of the block holding the page.

8.2.3. Dynamic Address Computation - Data Page Tables

This is similar to the steps above, except that the reference is to DIRECTORY(2).DATA_PAGES(1). The final address table is given in figure 69.
### Figure 69: Address Table for Data Page Tables

<table>
<thead>
<tr>
<th>Tag Span</th>
<th>Data Span</th>
<th>Index</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Directory</strong></td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Tag Pages</strong></td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td><strong>Block Nr</strong></td>
<td>4</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td><strong>Pge Length</strong></td>
<td>4</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td><strong>Clique</strong></td>
<td>4</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td><strong>Clique Length</strong></td>
<td>13</td>
<td>169</td>
<td>0</td>
</tr>
<tr>
<td><strong>Data Pages</strong></td>
<td>2</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td><strong>Block Nr</strong></td>
<td>8</td>
<td>104</td>
<td>0</td>
</tr>
<tr>
<td><strong>Pge Length</strong></td>
<td>8</td>
<td>104</td>
<td>0</td>
</tr>
<tr>
<td><strong>Clique Length</strong></td>
<td>8</td>
<td>104</td>
<td>0</td>
</tr>
</tbody>
</table>

**Top = 1**  **Bottom = 7**
Again, most of the components of this structure are factored, and the tags are stored in the DESCRIPTORS. The only tags that must be read are:

1 Tag for the DIRECTORY Component (in block 3).
2 Tags for the TAG_PAGES Component (in block 3).
4 Tags for the CLIQUES Component (in block 3).
1 Tag for the DATA_PAGES Component (in block 3).

Mapping the sites for these tags requires 3, 4, 5 and 6 references to the Directories Tag Page Table in block 1.

Adding up the DATASPAN values for the LEAF Components gives the total displacement in the DIRECTORY Data Structure to the beginning of the Tag Page Tables for the NODE (B-tree) Structure:

52+52+169+104+104+104 = 585 bits.

Mapping this site to the page table location requires 6 references to the directory Data Page Table in block 2 (4 PGE_LENGTH reads, 1 CLIQUE_LENG read and 1 BLOCK_NR read).
8.2.4. Dynamic Address Computation - B-tree Structure

Now that the Directory data (Page tables) have been located for the NODE Data structure, DAC can be performed to locate the data. This is very much like the operations involved in locating the directory data, except that the address table is smaller. The data specified is at NODE(4).KEYS_PER_NODE(2).KEY, and the final address table is in figure 70.
Figure 70: Address Table for B-tree

<table>
<thead>
<tr>
<th>NODE</th>
<th>TAGSPAN</th>
<th>DATASPN</th>
<th>INDEX</th>
<th>DONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>GU</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>2</td>
<td>GD</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>104</td>
<td>0</td>
<td>DD</td>
<td></td>
</tr>
</tbody>
</table>

TOP = 1  BOTTOM = 3
None of the components of this structure are Factored, so the tags that must be read are:

1 Tag for the NODE Component (in block 9).
4 Tags for the KEYS_PER_NODE Component (in block 9).
7 Tags for the KEY Component (6 in block 9 and 1 in block 11).

Mapping the sites for these tags requires 3, 3, and 11 references to the Tag Page Table in block 4.

8.2.5. Read the Data Using the SITE

Before the data can be read, the tag that gives its length must be read. This requires that a SITE be set up indexing to the eighth tag (7*8 bits) into the third clique. Mapping this site requires 6 references to the Tag Page Table for the B-tree (in block 4). One reference to block 11 reads the data.

To read the data, a SITE is set to all zeroes except for the PGE_INDEX, which is set to the DATASPN value for the one LEAF component (104). The SITE must then be mapped to the correct page. This requires 4 references to the Data
Page Table (in block 5). The result is a SITE that points to the data starting at bit 16 in block 19.

Now, a single reference to block 16 reads the data itself.

8.3. Performance Summary

The performance of the algorithms is measured in terms of the number of disk accesses required. The next section describes the method used to control and minimize the number of times that a reference to a block actually results in a disk access. The results of the previous example are summarized by enumerating the number of block references, more or less in the order that the references occur (figure 71).
To locate the Tag Page Table for the B-tree:

<table>
<thead>
<tr>
<th>Block</th>
<th>Contains</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Map</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Tags</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Map</td>
<td>5</td>
</tr>
</tbody>
</table>

To locate the Data Page Table for the B-tree:

<table>
<thead>
<tr>
<th>Block</th>
<th>Contains</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Map</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Tags</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Map</td>
<td>6</td>
</tr>
</tbody>
</table>

To Locate the data:

<table>
<thead>
<tr>
<th>Block</th>
<th>Contains</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Tags</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>Tags</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Map</td>
<td>17</td>
</tr>
</tbody>
</table>

To Get the Data Length:

<table>
<thead>
<tr>
<th>Block</th>
<th>Contains</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Map</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>Tags</td>
<td>1</td>
</tr>
</tbody>
</table>

To Read the data:

<table>
<thead>
<tr>
<th>Block</th>
<th>Contains</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Map</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>Data</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 71: Summary of Block References
The data can be grouped into a slightly different way, one that indicates where the work is being done. Grouped into the number of page references required for mapping sites, for reading tags for DAC, for reading tags for lengths and for actual reading of data, one can easily determine where improvements in the algorithms might result in performance improvements (Figure 72).
Number of Block References for:

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td>74</td>
</tr>
<tr>
<td>Reading Tags in AC</td>
<td>24</td>
</tr>
<tr>
<td>Reading Tags for Len.</td>
<td>1</td>
</tr>
<tr>
<td>Reading the Data</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

Figure 72: Block References by Function
If this was the number of blocks that actually had to be read from disk, it would be, without question, unreasonably large. However, it only represents references to eight disk blocks. Algorithms will be described in the next chapter that allow multiple references to a page while only reading the corresponding disk clock once. Obviously, reducing the number of page references required for mapping and AC are profitable areas to consider to improve the performance of the system.
9. PAGE MANAGEMENT AND PERFORMANCE IMPROVEMENTS

The preceding chapters 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 also possible to improve the performance of these algorithms (and, incidentally, to simplify their analysis) by making several modifications. This chapter 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.

9.1. Page Management

The data accessing algorithms previously described (GDATA, RDATA, IDATA and DDATA) 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.
In the example in the previous chapter, there were usually several references to each block. In fact, more than half (59 out of 100) were to blocks 1, 2 and 3. To take advantage of this fact, several "buffers" are allocated, each providing space for one block. Then, whenever reference is made to a block, if the block is not already in a buffer, one of the buffers is chosen, and the block is read, replacing any block already in that buffer. If there was a modified block in the buffer, it is written back to the file before the new block is read. This is simply a small scale version of the classical demand paging concept used in operating systems for virtual storage [BRINP73]. The buffer to be used is chosen using a simple Least Recently Used (LRU) replacement algorithm. In the example in the previous chapter, only eight blocks were actually used. Thus, eight buffers would have been sufficient to result in only eight reads from the disk.

9.2. Performance Improvement

There are several modifications to the basic DAC and page management algorithms that will 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, resulted in notable performance improvements (discussed later).

9.2.1. 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 first three pages in the example in chapter 8). 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.

Notice that 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). In the preceding example, this would further reduce the number of disk reads to five.

9.2.2. 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 (refer to the example in chapter 8). 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).

9.2.3. 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_MULTI 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.

9.2.4 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.

9.2.5. 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. In the example in chapter 8, it was fully 74%
of the page references. 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.
9.2.6. Splitting Pages

In the implementation described here, when a page fills a block and more data must be inserted, the page is split between the current block and another, newly allocated block. The splitting is done by simply placing half of the tags from each clique into each block. A better splitting algorithm would recognize data structure occurrence boundaries, and attempt to split there (Cook calls these "Address Cuts"). This would keep tags more closely associated with the tags of the parent and descendant substructure occurrences.

This modification was not made in the implemented system.
10. 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 chapter also make the analysis somewhat more tractable.

This chapter will present an analysis of several aspects of the performance of the system. Following Wiederhold's example [WIEDG77], we will consider one of the performance measures to be the (disk) storage required to hold the tree. The cost (in disk blocks transferred) to search the B-tree (and read the indexed data) will be discussed in some detail.

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 WIEDG77), 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.
10.1. Notation

The following notation will be used throughout this chapter for the basic parameters. Other symbols will be defined as needed.

\( m \) - 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 (\( Bk \times Ld = \text{Average data on a page} \)).
\( \text{LEN_LEN} \) - The length of a length entry in a Page Table (ex. CLIQUE_LENG).
\( \text{LEN_NR} \) - The length of a BLOCK_NR entry in a Page Table.
10.2. Storage Requirements

There are really three data structures being stored, the B-tree, the Indexed Data and the Directory. Thus, the total storage required is simply:

\[
\text{Total Storage} = \text{Storage for B-tree} + \text{Storage for Indexed Data} + \text{Storage for Directory}
\]

Notice that this assigns the Page Tables that describe a data structure to the Directory, rather than to the data structure itself. This is a direct consequence of the fact that the Directory is just another data structure, and is best treated as an entity unto itself.

The storage for a given data structure can be divided into two parts, the tags that describe the data, and the data itself. So, for each of the B-tree, the Indexed data and the Directory:

\[
\text{Storage Required} = \text{Storage for Tags} + \text{Storage for Data}
\]
The storage for the Tags is simply the number of tags times the size of the tags.

For the B-tree tags:

If the Keys are factored (only the KEYS_PER_NODE component has tags):

\[ \text{Storage for Tags for B-tree} = T_{kpn} \times N_n \]

If the Keys are unfactored:

\[ \text{Storage for Tags for B-tree} = T_{kpn} \times N_n + T_{key} \times N_k \]

For the Indexed Data:

If the Indexed Data is factored (there are no tags):

\[ \text{Storage for Tags for Indexed Data} = 0 \]

If the Indexed Data is unfactored:

\[ \text{Storage for Tags for Indexed Data} = T_{data} \times N_d \]
For the Directory, since it uses Locally Factored Tags in the CLIQUES Component, there are a fixed number of tags, no matter how big the B-tree grows. The fixed storage required for these tags is:

\[
\text{Storage for Tags for Directory} = 3 \times \text{Ttp} + 3 \times \text{Tcl} + 3 \times \text{Tdp}
\]

There are actually fewer tags when the Indexed Data is fixed length (Factored), but the Directory tags use such a small amount of space compared to any significant sized B-tree, that the value just derived, which is the maximum possible size, will be used.

The storage for the data itself is very simply stated for the B-tree and the Indexed Data. For the B-tree, the storage required is:

\[
\text{Storage for data for B-tree} = Nk \times K
\]

For the Indexed Data:

\[
\text{Storage for data for Indexed Data} = Nd \times D
\]
For the Directory, it is somewhat more complex, as it is a function of the storage required for the B-tree, the Indexed Data and the Directory itself. This last fact, that the Directories size depends on the size of the Directory, makes the storage requirement a recursive function:

Storage for Data for Directory =

Storage for Page Tables for B-tree +
Storage for Page Tables for Indexed Data +
Storage for Page Tables for Directory

There are two sets of page tables (PT) for each data structure, the Tag Page Tables and the Data Page Tables. Therefore, the storage for the page tables for a structure is:

Storage for Page Tables =

Storage for Tag PT + Storage for Data PT

Where: Storage for Tag PT =

Number of Tag Pages * Size of Tag PT entry

and: Storage for Data PT =

Number of Data Pages * Size of Data PT Entry
The number of Page Table entries is exactly the number of pages used to store the corresponding data (tags or data). The size of a Page Table entry (derived later) is a function of the number of cliques and the size of the values in the table.

For the B-tree:

Number of Tag Pages for B-tree =

\[
\frac{\text{Storage for Tags for B-tree}}{\text{Ld} \times \text{Bk}}
\]

For the Indexed Data:

Number of Tag Pages for Indexed Data =

\[
\frac{\text{Storage for Tags for Indexed Data}}{\text{Ld} \times \text{Bk}}
\]

Remember that the Directory Tags use a small, fixed amount of space, so:

Number of Tag Pages for Directory = 1

The number of Data Pages is simple for the B-tree and Indexed Data, but is fairly complex for the Directory.
For the B-tree:

Number of Data Pages for B-tree =

\[
\frac{\text{Storage for Data for B-tree}}{Ld \times Bk}
\]

For the Indexed Data:

Number of Data Pages for Indexed Data =

\[
\frac{\text{Storage for Data for Indexed Data}}{Ld \times Bk}
\]

The Directory data is actually the Page Tables themselves. The Page Tables for the Directory, for the B-tree and for the Indexed Data do not share pages (See the example). This means that the number of pages for the
Directory data is the sum of the number of pages required to store the page tables for all of the structures. Thus:

Number of Data Pages for the Directory =

\[
\begin{align*}
\text{Storage for Tag PT for B-tree} & \quad + \\
\text{Storage for Tag PT for Indexed Data} & \quad + \\
\text{Storage for Tag PT for Directory} & \quad + \\
\text{Storage for Data PT for B-tree} & \quad + \\
\text{Storage for Data PT for Indexed Data} & \quad + \\
\text{Storage for Data PT for Directory} & \quad + \\
\end{align*}
\]

Since, by assumption, the Page Tables for the Directory fit on one page:

\[
\begin{align*}
\text{Storage for Tag PT for Directory} & = 1 \\
\text{Storage for Data PT for Directory} & = 1
\end{align*}
\]
This terminates the recursion in the definition of the amount of space required to store the Directory data.

The last values required are the sizes of the individual page table entries. These are always one BLOCK_NR, one PGE_LENGTH and one CLIQUE_LENG for each clique (always one for Data Pages). The number of cliques depends on whether the data is factored or not (only KEYS and the Indexed Data can be factored or unfactored; the format of the Directory is fixed). Thus:

For the B-tree:

If the Keys are factored:

Size of Tag PT entry = LEN_NR + LEN_LEN + LEN_LEN

And

Size of Data PT entry = LEN_NR + LEN_LEN + LEN_LEN

If the keys are unfactored:

Size of Tag PT entry = LEN_NR + LEN_LEN + 2 * LEN_LEN

And

Size of Data PT entry = LEN_NR + LEN_LEN + LEN_LEN
For the Indexed Data:

If the Indexed Data is factored:

Size of Tag PT entry = 0 (No cliques)
And
Size of Data PT entry = LEN_NR + LEN_LEN + LEN_LEN

If the Indexed Data is unfactored:

Size of Tag PT entry = LEN_NR + LEN_LEN + LEN_LEN
And
Size of Data PT entry = LEN_NR + LEN_LEN + LEN_LEN

For the Directory:

Size of Tag PT entry = LEN_NR + LEN_LEN + 3 * LEN_LEN
And
Size of Data PT entry = LEN_NR + LEN_LEN + LEN_LEN

10.3. Runtime Performance

The cost to search a B-tree and read the associated Indexed Data entry will be derived in this section. In
addition to the parameters described in the Notation section, the following notation will be used:

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.

These 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.

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.

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:

\[
\text{LAST}_n = \sum_{i=1}^{x-1} \left( \text{# of level } i \text{ nodes} \right) + \frac{\text{# of level } x \text{ nodes}}{2}
\]

1. The number of level 1 nodes is 1.

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

\[
\sum_{j=1}^{\text{number of level } i-1 \text{ nodes}} \left( 1+N_{\text{keys}}(\text{First}_\text{prev} + j - 1) \right)
\]

There is no closed form solution for these equations, so the algorithm in figure 73 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 73: Computation of LASTn
LASTk is simply the middle key of node LASTn, so:

\[ \text{LASTk} = \frac{\text{N}\_\text{keys (LASTn)}}{2} \]

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:

\[ \text{LASTo} = \text{LASTk} + \sum_{j=1}^{(\text{LASTn} - 1)} \text{N}\_\text{KEYS}(j) \]

\[ = (\text{Equation 1}) - \text{LASTn} \]

The analysis will be 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, it is important to 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 equation 1, 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.

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). Without the restart point, this would require reading every tag for the KEYS_PER_NODE component from the first through tag number LASTn. It would also require reading every tag for the KEY component from the first through the tag corresponding to the key being read (LASTo). 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.
The DAC procedures that read tags (GEN_MULT and GEN_DIV) both use "site relative" addressing. That is, after the first tag is read, the next tag is read by setting SITE.INDEX to the length of the tag just read and calling GDATA. This means that the site is usually mapped using information contained in the SITE. The only time that Directory data is needed is when the tag reference reaches a page boundry in the tag pages. Then, one Tag Page Table entry is read from the Page Table, and the process continues.

The result of this is that whenever two tags are read in succession by GEN_MULT or GEN_DIV (all tags required for a component are read this way), no other pages are referenced between reading of tags. So it is guaranteed that the block will still be in memory, and no page fault can occur.

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:

\[
\text{Tag pages read} = \frac{\text{LASTo} \times \text{Tkey}}{\text{Ld} \times \text{Bk}} + \frac{\text{LASTn} \times \text{Tkpn}}{\text{Ld} \times \text{Bk}}
\]

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:

\[
x \times \frac{a}{2} \times K
\]

\[
\frac{B\text{-tree Data Pages Read}}{Ld \times Bk} = x \times \frac{a}{2} \times K
\]

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

\[
N_{tpt} = \frac{L_d \times B_k}{\text{LEN}_\text{NR} + 3 \times \text{LEN}_\text{NR}}
\]

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

\[
N_t = \frac{L_d \times B_k}{T_kpn \times N_n + T_key \times N_k}
\]

tags. This means that one Tag Page Table page is sufficient \( N_{tpt} \times N_t \) tag pages for the B-tree. Both \( N_{tpt} \) and \( N_t \) 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:

\[
\text{Tag Page Table Pages read} = 1
\]

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_LEN 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 (eg., 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

\[
Ndpt = \frac{Ld \times Bk}{LEN_NR + 2 \times LEN_NR}
\]

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:

\[
\text{Number of B-tree Data Pages} = \frac{Ndpt}{Ndpt}
\]

So:

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

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

\[
\frac{\left( \frac{a}{2} \right) \left( \frac{1}{2} \right)}{Ndpt} \]

where \( a = \text{average number of keys in a node} \).
10.4. An Example

The equations just derived will be applied to the sample B-tree from chapter 8 with the following parameters:

\[ m = 4 \]
\[ K = 15 \text{ (bits)} \]
\[ D = 61 \text{ (bits)} \]
\[ N_n = 10 \text{ (nodes)} \]
\[ N_k = 24 \text{ (keys)} \]
\[ N_d = 24 \text{ (Records)} \]
\[ T_{kpn} = 10 \text{ (bits)} \]
\[ T_{key} = 8 \text{ (bits)} \]
\[ T_{data} = 8 \text{ (bits)} \]
\[ T_{tp} = 12 \text{ (bits)} \]
\[ T_{cl} = 4 \text{ (bits)} \]
\[ T_{dp} = 12 \text{ (bits)} \]
\[ B_k = 320 \text{ (bits)} \]
\[ L_d = 0.37 \]
10.4.1. Storage Requirement Computation

The storage for the various pieces is:

A = Tags for B-tree
   = Tkpn*Nn + Tkey*Nkey
   = 10*10 + 8*24
   = 292 bits

B = Tags for Indexed Data
   = Tdata*Nd
   = 8*24
   = 192 bits

C = Tags for the Directory
   = 3*(Ttp+Tcl+Tdb)
   = 3*(12+4+12)
   = 84 bits

D = Data for B-tree
   = Nk * K
   = 24 * 15
   = 360 bits
E = Data for Indexed Data
    = Nd * D
    = 24 * 61
    = 1464 bits

F = Data for the Directory (Computed below)

To compute the storage for the Directory data:

L = Number of Tag Pages for B-tree
    = \left\lceil \frac{A}{(L_d*B_k)} \right\rceil
    = \left\lceil \frac{292}{118} \right\rceil
    = 3 pages

M = Number of Tag Pages for Indexed Data
    = \left\lceil \frac{B}{(L_d*B_k)} \right\rceil
    = \left\lceil \frac{192}{118} \right\rceil
    = 2 pages

N = Number of Data Pages for B-tree
    = \left\lceil \frac{D}{(L_d*B_k)} \right\rceil
    = \left\lceil \frac{360}{118} \right\rceil
    = 4 pages
O = Number of Data Pages for Indexed Data
   = \left\lceil \frac{E}{(L_d \times B_k)} \right\rceil 
   = \left\lceil \frac{1464}{118} \right\rceil 
   = 13\;\text{pages}

P = Storage for Tag PT for B-tree
   = L \times (\text{LEN}_{NR} + \text{LEN}_{LEN} + 2 \times \text{LEN}_{LEN})
   = 3 \times 52 
   = 156\;\text{bits}

Q = Storage for Tag PT for Indexed Data
   = M \times (\text{LEN}_{NR} + \text{LEN}_{LEN} + \text{LEN}_{LEN})
   = 2 \times 39 
   = 78\;\text{bits}

R = Storage for Tag PT for Directory
   = 1 \times (\text{LEN}_{NR} + \text{LEN}_{LEN} + 3 \times \text{LEN}_{LEN})
   = 65\;\text{bits}
S = Storage for Data PT for B-tree
    = 0*(LEN_NR+LEN_LEN+LEN_LEN)
    = 4*39
    = 156 bits

T = Storage for Data PT for Indexed Data
    = P*(LEN_NR+LEN_LEN+LEN_LEN)
    = 13*39
    = 507 bits

Now, the number of pages for the Data Page Table and for
the Tag Page Table for the Directory is 1 each, by
definition. So the number of data pages for the Directory
is:

\[
\begin{align*}
\left[ \frac{P}{(Ld*Bk)} \right] &+ \left[ \frac{Q}{(Ld*Bk)} \right] +1 &+ \left[ \frac{R}{(Ld*Bk)} \right] &+ \left[ \frac{S}{(Ld*Bk)} \right] &+ 1 \\
    & = 2 &+ 1 &+ 1 &+ 2 &+ 5 &+ 1 \\
    & = 12 \text{ pages}
\end{align*}
\]

and the Storage for the Page Table for the Directory is:

U = 12*(LEN_NR+LEN_NR+LEN_LEN)
    = 12*39
    = 468 bits.
Now, the total storage for the Directory Data (F) is:

\[ F = P + Q + R + S + T + U \]
\[ = 156 + 78 + 65 + 156 + 507 + 468 \]
\[ = 1430 \text{ Bits} \]

and the total storage for the entire tree and related structures is

\[ A + B + C + D + E + F \]
\[ = 292 + 192 + 84 + 360 + 1468 + 1430 \]
\[ = 3826 \text{ Bits.} \]

This compares favorably (within 3 percent) with the actual value for the example structure of 3937 bits.

10.4.2. Access Cost Computation

To find the key and data record "55":

LASTn=10
LASTk=2
LASTo=16
To read the B-tree tags:

\[
\text{Number of pages} = \frac{\text{LASTo} \times \text{Tkey}}{\text{Ld} \times \text{Bk}} + \frac{\text{LASTn} \times \text{Tkpn}}{\text{Ld} \times \text{Bk}}
\]

\[
= \frac{16 \times 8}{118} + \frac{10 \times 10}{118}
\]

\[
= 1 + 1
\]

\[
= 2 \text{ pages}
\]

To read the Key:

\[
\text{Number of pages} = \frac{x \times (a/2) \times \text{K}}{\text{Ld} \times \text{Bk}}
\]

\[
= \frac{3 \times 2.5 \times 0.5 \times 15}{118}
\]

\[
= 1 \text{ page.}
\]

To read the Directory data:

\[
\text{Number of pages} = (x - 2) \left( \frac{a}{2} \right) \left( \frac{1}{2} \right) \frac{\text{Nr Data Pages}}{\text{Ndpt}}
\]

\[
= \frac{2.5 \times 1 \times 4}{2 \times 2 \times 3}
\]

\[
= 1 \text{ page.}
\]

So the total number of pages read is 4. In the example of reading a key, the number of pages actually touched was 8. This included the three pinned pages, which will not be counted as a read. So the number of pages read was
really 5. That example was not full search of the B-tree, but rather the cost to read a key on the last level. However, since reading the first and second level keys would have required reading most of the same pages, this computed cost is within one or two pages of the actual example. The artificially small example means that many of the assumptions made in the analysis were not actually true, so this result is reasonable.
11. CONCLUSIONS

In this thesis, a complete implementation of Dynamic Address Computation (DAC) as described by Cook [COOKT77] was presented in detail. The thesis 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 chapter, 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 chapter contains some comments about the implementation, lessons learned, and possible future directions.

11.1. It Worked

Cook had described the DAC algorithms and suggested that the B-tree implementation was possible. This work
demonstrates the correctness of his suggestion. The entire DAC system was implemented and the B-tree algorithms successfully modified to run using DAC.

11.2. The B-tree Implementation was Easy

The B-tree algorithms and their implementation took about 1% of the total effort involved in this work. The use of the ragged array model makes the B-tree algorithms very simple, since the complications involved in keeping track of pointers and associating them with the correct keys are eliminated. Most of the actual code and effort went into the DAC mechanism. If an operating system that directly supported DAC (or a some other ragged array model) is available, the B-tree implementation is almost trivial.

11.3. 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 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.

11.4. 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).

11.5. It Needs more Work

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 [KNUTD73]. 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 chapter 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 in chapter 9) should result in sizeable gains.

11.6. 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.
Appendix A: GLOSSARY

The following are either new terms, therefore unfamiliar to most readers, or general terms that are used in a particular way in this paper, so they need to be defined.

Address Computation:

dynamically deriving the address on a datum in a file using the file structure definition (schema) and datum length data (tags). (SEE ALSO - Instance Equation, Schema, Tag)

Block:

The unit of transfer between disk and main memory. A fixed size area on disk. (SEE ALSO - Page)

Clique:

A collection of all of the tags for a single, unfactored component. (SEE ALSO - Tag, Component, Locally Factored Component, Unfactored Component)
Component:
A named element of a data structure, corresponding to one row in the data structure definition. For example:

```
01 DATA_NODES repeats max(max_nodes),
  02 DATA   leaf max(max_data);
```
defines a single data structure with two components, named DATA_NODES and DATA. The components may be referenced by name or by number. In this case, component number one is DATA_NODES and component number two is DATA.

DATASSPAN:
A "data distance", given as the number of substructure occurrences or the number of bits of data, for a particular component, preceding a given piece of data. (SEE ALSO - TAGSPAN)

Descriptor:
An element of the SCHEMA that describes a component. Included in the descriptor is type (REP, LEAF), format (UNF, FACT, LFACT), the size of a tag for the component, the units of the tags, the clique number (if UNF or LFACT), the tag value for FACT components and an identifier of the parent component.
FACT:
An abbreviation for factored, one of the component formats.

Factored Component:
a component whose tags will always have the same value, so that this value can be "factored out" into a single tag, stored in the descriptor for the component. (SEE ALSO - Unfactored Component, UNF, Locally Factored Component, LFACT)

Indexed Data:
The data stored in a database that is the ultimate target of all references to the database. In most database management systems, this is the records that hold all of the data stored in the database. (SEE ALSO - Key)
Instance Equation:
The fundamental relationship in Dynamic Address Computation, specifying the relationship between TAGSPAN and DATASPAN for a component. For component $k$, the Instance Equation is:

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

(SEE ALSO - DATASPAN, TAGSPAN)

Key:
A value (such as the value of a field in a record) that is to be stored in an index (ex., B-tree) to allow fast access to the indexed data (records) associated with that value.

Leaf:
a term used to refer to the data holding components of a data structure, as opposed to the substructure defining components. (SEE ALSO - REP)
LFACT:

An abbreviation for Locally Factored, one of the component formats.

Locally Factored Component:

A component whose tags are always the same value within an occurrence of the containing substructure, but which can vary between occurrences of that substructure. (SEE ALSO - Factored Component, FACT, Unfactored Component, UNF)

Page:

A logical entity referring to the data in a single block. A page can be from 0 to Block_size bits long. There is one page in a block, and any unused bits in the block are available to expand the page. (SEE ALSO - Block, Page Table)

Page Table:

A table used to map an offset into a data structure into a block number and an offset within that block. The page table gives, for each page, the block number, the size of the page in that block, and information about how the page space has been allocated among the cliques. (SEE ALSO - Block, Clique, Page)
REP:
An abbreviation for Repeats, one of the component types.

Repeats Component:
A term used to describe a non-leaf component in a data structure. A REP component is used to group the components below it into a single, named substructure. (SEE ALSO - Leaf Component)

Schema:
The description of the structure of a data structure. The schema is made up mostly of the descriptors for the components of the data structure. (SEE ALSO - Descriptor)

SITE:
A temporary data structure that gives the address of a piece of data. It contains the Block number and the offset within the block where the data is located. It also contains data used in mapping of displacements to block numbers and offsets. The SITE is produced by using a page table to map a displacement (from the beginning of a data structure or from the current SITE position) to the block number and offset.
Tag:
A number that defines the number of bits that make up an occurrence of a LEAF component, or the number of occurrences of the subordinate components that make up an occurrence of a sub-structure (REP Component). There is one tag for a Factored Component and one tag for each occurrence of an Unfactored Component. (SEE ALSO — Clique, Factored Component, Unfactored Component, Locally Factored Component)

TAGSPAN:
The number of tags used in the computation of a DATASPN. (SEE ALSO — DATASPN)

UNF:
An abbreviation for unfactored, one of the component formats.

Unfactored Component:
A component whose tag values may change between occurrences of the substructures and data items that compose the substructure. (SEE ALSO — Factored Component, FACT, Locally Factored Component, LFACT)
The vita has been removed from the scanned document