Bee Tree

by Frederick L. Hiltz, Stage Manager

M databases are faster and more compact than others. How does this happen? Most M implementations store global variables in the fixed-size blocks of a disk with a tree structure called the *B tree*.

M programmers know trees. We describe them in the customary computer-science orientation, root at the top and leaves at the bottom, using these terms:

> A *node* is one global variable identified by a *key* composed of the global name and subscripts and optionally containing data. The global name alone identifies the *root*, from which all other nodes descend. Every node has a *depth*, the number of subscripts in the key. Its *siblings* are other nodes at the same depth, its *children* are its immediate descendant nodes, and its *parent* is the node from which it descends.

Three characteristics define a B tree:

- A node may have many children.
- All data are stored in the leaves. Higher nodes contain only keys and links to other nodes.
- The depth of all leaves is the same.

M globals are definitely not B trees: they may contain data at any level, and leaf nodes may have different depths. The B tree is, however, an excellent structure for the disk blocks that store M globals. The rest of this story is about the embedding of an M global's nodes into a B tree of disk blocks.

Let us begin building a B tree the way you would begin a pyramid, with the blocks on the bottom, the leaves that hold the data. One or more nodes of an M global occupy a block; thus the size of the block limits the size of a node. Each node consists of a key (derived from its global name and subscripts) plus its data, and the nodes dwell in collating sequence. In addition to these nodes, the block contains a pointer to its sibling where the sequence continues.

The leaf blocks form a sequential file of nodes that a program may traverse with $QUERY. Random access, however, requires a way to find nodes anywhere in the sequential file. Blocks at the next higher level, called index blocks, accomplish this. Each index block contains *pointer nodes* in collating sequence. A pointer node comprises the key of the *first* node in a data block plus the number of that data block. Index blocks also contain pointers to their siblings.

The index blocks, therefore, form a sequential file of pointers that is much shorter than the file of data blocks (a typical index block contains pointers to 20 to 50 data blocks). One could traverse the index file quickly to find the number of the block that holds a data node, and many database products do exactly this, calling it the indexed sequential access method (ISAM).

Traversing the index blocks would be slow for large files, so the B tree adds another layer of index blocks above, containing pointer nodes to the index blocks below. At this point, a 200—Mbyte global stored in 4—Kbyte blocks might look like this:

```
        <- 40 index blocks ->
      <— 1250 index blocks —>
<———— 50,000 data blocks ————>
```

To complete the B tree, add more levels of index blocks, each about 1/40th the size of the level below, until the top level contains just one block, the root. Now one can locate any node of our 200-Mbyte global by reading and searching just four disk blocks:

1. Read the root block.

2. Search the block for the key that follows the desired key. Then back up one node, get the pointer to (block number of) its child, and read that block.

3. Perform step 2 twice more, ending by reading the data block that contains the desired node.

4. Search the data block for the key of the desired node.

To insert a node, search in the same manner for the node that follows the new node, and insert the new node into the data block. If the new node is first in that block, visit its parent index block and replace the former first node's key with the new node's key. The new key might be first in the index block; then *its* parent block requires the same substitution of the new key and so on until reaching the root.

What if the data block does not have room for the new node?

1. Split the block by finding a new unused block and moving the last half of the nodes into it.

2. Visit the parent block and insert a pointer node containing the number of the new block and the key of its first node.

3. If the index block does not have room for the new pointer node, apply steps 1 and 2 to it and so on until reaching the root.

After many insertions the root block must be split. The tree cannot have two roots, so one creates a new parent block and inserts pointer nodes to the pair; the tree grows one level higher.

Deleting a node is the converse of inserting a node. When adjacent sibling blocks gain sufficient empty space, they are merged, and the pointer to the now-unused block is deleted from the parent node. After many deletions the root has just one child; the root may be discarded, the child becomes the new root, and the tree shrinks one level.

Knuth demonstrates that, after many random insertions and deletions, blocks are 60% to 70% full on average.[1]

This completes the textbook description of the B tree. Practical implementations apply several techniques to improve speed and storage efficiency: Clever derivation of a key from the global name and subscripts permits rapid comparison of keys while preserving the M collating sequence. Adjacent keys contain many common leading characters; "key compression" eliminates the redundant characters among those that occupy the same block. Some trailing characters are not needed to distinguish keys in index blocks; they may be omitted.

Index blocks may differ in size from data blocks, and they may reside in separate files of the operating system. When a B tree is built in collating sequence, as from an ordered sequential file, blocks may be close to 100% full. The downward pointers and sibling pointers provide redundant descriptions of the tree structure that may be used for checking and restoring the integrity of the database after an accident. A cache of disk blocks keeps most high-level index blocks and recently used data blocks in memory, dramatically increasing speed.

Knuth provides the classical textbook treatment of several variations on the B tree including derivations of their dynamic behavior.[1] Lewkowicz presents an excellent description of the B tree in the M environment.[2] ■

Endnotes

1. D. E. Knuth, *The Art of Computer Programming* (Reading, MA: Addison-Wesley, 1981).
2. J. M. Lewkowicz, *The Complete MUMPS* (Englewood Cliffs, New Jersey: Prentice Hall, 1989).

Frederick L. Hiltz, Ph.D., develops medical information system software at Brigham and Women's Hospital, Boston, Massachusetts.

M Technology: Integrating the Best

M TECHNOLOGY ASSOCIATION EUROPE
20th ANNUAL MEETING
BARCELONA 8-10 NOVEMBER 1995