B-TREES RE-EXAMINED

by

Gerald Held and Michael Stonebraker

Memorandum No, ERL-M528

2 July 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

B-trees Re-examined

by

Gerald Held and Michael Stonebraker

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

Key Words and Phrases: B-tree, directory, static directory, dynamic directory, index sequential access method

CR Categories: 3.70, 3.73, 3.74, 4.33, 4.34

The B-tree [1] has been receiving considerable attention as a storage structure for certain files on paged secondary storage devices. Such files include those consisting of a set of records each of which has an identifying portion called a KEY. Access to the file is desired both randomly (by requesting the record corresponding to a given key) and sequentially (in collating sequence by key value).

In this note we briefly explain B-trees, several of their variants and an alternate static directory structure. Then we indicate some potential problems which a B-tree implementation must overcome in a multi-user data base environment that do not arise in static directory structures. These problems include a possible performance penalty.

Research sponsored by the National Science Foundation Grant DCR75-03839.

B-TREES

Basically, a B-tree is a balanced tree with between k+1 and 2k+1 sons for any given node. The parameter k is determined by the page size and data characteristics. An example of a B-tree for k=1 is shown in figure 1. Here, space exists on each block (or page) for two data records and three pointers. (In the figure we only indicate the key portion of the record.) Note that the records are in collating sequence if the tree is scanned in post-order. Note also that the number of page accesses required to reference any given record is logarithmic in the number of data records.

The major advantage of this structure is that the tree can be kept balanced during insertions and deletions with a known (and small) worst case update cost. Hence, a small worst case search time is always guaranteed. For a detailed discussion of the B-tree maintenance algorithms, see [1].

For example, the record with key of ALL can be added to the tree with only three page accesses and stored in the empty space on the page labeled A. However, the addition of a record with key of ELF will not fit on the page labeled B and hence, B is split into two pages and a record moved to a higher level node. This, in turn, causes the page labelled C to split and the final structure that results is the balanced tree shown in figure 2. Note clearly that several records have been moved as a result of this

insertion. Such necessary address modifications will be shown to be potentially troublesome.

The B-tree is one example of a class of storage structures which we call a "dynamic directory" because it is dynamically reorganized during updates to provide a balanced search tree (or directory) at all times.

There are other variations of dynamic directories. One variant is discussed in [2] and offer obvious advantages over B-trees. The original proposal placed entire records in directory pages. In fact, placing only keys in the directory increases k and reduces the height of the tree. Hence, data pages can be accessed with fewer retrievals from secondary storage. Also, the minimum number of records on a page can be increased above k+1 by a spillover technique to adjacent pages in order to avoid page splits. This idea underlies B*-trees also discussed in [2]. VSAM [3] is another variant of dynamic directories. In the sequel we will be solely concerned with those dynamic structures having only keys in the directory levels.

STATIC DIRECTORY STRUCTURES

Storage structures for which the index levels are not changed dynamically, such as ISAM [4], will be termed "static directories". Figure 3 indicates one such static directory structure for the same data used in figure 1. This directory structure is

implemented as one access method in INGRES [5]. Four important points should be noted:

1) The index levels are formed by recording the high key on each data page.

2) Once formed, the index levels are NOT dynamically altered (in contrast to a dynamic directory).

3) As a result of 2), only one pointer per index page is required. In our example, the three data pages pointed to can be logically (or physically) contiguous. Because of this pointer suppression we have assumed three keys fit on a direcory page instead of the two in a dynamic directory.

4) Additions to the structure of figure 3 are handled by chaining into overflow areas. The addition of a record for ALL would cause page A to split, in which case an overflow page would be allocated and chained onto page A as noted in figure 4. Note clearly that existing records are not moved and that the collating sequence within a primary page and its overflow pages is not maintained. If records must be kept in collating sequence, a chain of pointers through the records can easily be supported. Reusing space of deleted records is also easily allowed but the mechanism is not considered in this note.

As a storage structure for a single isolated data file, dynamic directories appear very useful. However, this class of

structures has recently been receiving much attention as a candidate for the basic storage structure in several data base management systems. In a data base environment we feel there are three important points to be considered before adopting such a structure.

POINTS OF COMPARISON BETWEEN DYNAMIC AND STATIC DIRECTORIES

1. Secondary Indices

If it becomes necessary to access the file on some portion of the record that is not used as a directory key, a sequential scan of the entire file may be required. In the example data base, such a scan is required to access all records with keys ending in Y. This problem is often alleviated by using secondary indices (inversions on attributes) [6].

In this case a secondary file is maintained that contains pairs of attribute values and pointers to records in the primary file which have that value. Figure 5 gives an example of a secondary index that might be used in conjunction with the structure of figure 1. In that figure ->BAD indicates a pointer to the record for BAD in the primary file.

Independant of the storage structure chosen for the data file, there is an overhead in maintaining such indices during updates to the primary file. When the data file is updated (by adding, deleting, or changing records) then the indices must be updated

to reflect newly added, deleted, or changed values of the inverted attributes. However, if the data file is a dynamic directory, in additon to these updates, other secondary updates may be generated because of the dynamic reorganization. These additional updates occur when an update operation causes a data page to split or merge (as happened during the second insertion into the B-tree of figure 1). In such cases records must be moved to new pages and be assigned new logical (or physical) page addresses. Every record which is assigned a new address will require an update to be generated for each existing secondary index. Such additional updates could be avoided if the secondary index used the primary key of the data record instead of its address as a pointer. This approach, however, would require primary key decoding through the directory for each access using a secondary index.

2. Concurrency

The second problem with dynamic structures occurs when concurrent processes use the same file. Suppose two processes are simultaneously accessing a dynamic directory; one inserting a record and one performing a scan over a portion of the tree. Suppose further that the scanning process is partway through a page when the updating process causes that page to be split by the insertion. This rearrangement will leave the scanning process pointing to a wrong (or non existent) record unless the updating

process alters the scan pointer of all other processes in a non-trivial way.

Other concurrency problems arise when two processes concurrently update the same B*-tree. Suppose that the two processes are adding a record to two adjacent pages in the tree and suppose both pages are full. Each process must lock the page it is updating since it will be altered. Then it must examine the two adjacent pages to see if records can be spilled over into them to avoid a split. However, the adjacent page is locked and each process is requesting access to the page the other has locked. Clearly, this deadlock situation must be recognized and broken.

3. Directory Height

The third problem with dynamic structures involves the height of the directory tree. Because pages are split on the fly in a dynamic directory, explicit pointers to data pages must be present in the higher levels of a dynamic directory. Notice, for example, that space must be left for three pointers on each page of figure 1. These pointers consume space and limit the value of k that can be attained.

These problems can all be avoided in static directory structures. A static directory structure can have the property that records are never moved (if chaining between records in the primary and overflow areas is allowed or if the records on a given page and

its overflow pages are not kept in collating sequence as in figure 4). If this is the case, pointers can be safely used in secondary indices. Moreover, the directory is static; therefore, an updating process need only lock the page it is modifying and no others. Also, since records are not moved, there is no danger of a scanning process pointing to a non-existent record. In addition, pointers in the directory can be easily suppressed thus increasing the fanout possible (often by as much as a factor of two). Often this will save a level of the directory as a subsequent example will illustrate. Lastly, because the directory is static, elaborate coding and interpolation schemes are often possible which are precluded by dynamic directories. One such scheme is indicated in [7]. Such coding schemes may well save a level of the directory.

Of course, the primary disadvantage of a static directory structure is the necessity of periodic reorganization when the overflow area becomes highly utilized.

In order to further compare performance, we present two simple examples. Suppose pointers and keys are both four bytes in length and that the page size is 512 bytes (the size used in the UNIX [8] operating system on top of which INGRES is implemented). In this case a B-tree (assuming only keys are present in the index levels) has nodes with between 32 and 63 sons. On the other hand, the number of sons for an index structure of the form of

figure 3 is 127. The following table indicates the height of the tree for various sizes of the primary data to be indexed. In both cases we assume the index nodes are completely full.

number of data pages	height of a tree with a static directory
2-127	2
128-16,129	3
16,130-2,048,383	4

number of data pages	height of a tree with a dynamic directory
2-63	2
64-3968	3
3969-250,047	4
250,048-15,752,961	5

A Comparison of Tree Heights

Table 1

The important point to note from Table 1 is that a static directory will save a level in the tree for any file between 3969 and 16,129 pages (roughly 2-15 million bytes) and also for files over 125 million bytes. For such files, a static directory requires one less disk read than a dynamic directory for each retrieval request. Of course, this example is sensitive to the page size,

key size and pointer size chosen. However, a level is saved in many situations.

Our final example involves an 8,000 page file with the above directory characteristics. Suppose further that four records fit on a page and that 32,000 inserts are performed. Lastly, suppose inserts are done such that the effect on the structure of figure 3 is to add an overflow page to each primary page. The effect on a comparable B-tree is to split each data page.

Each B-tree insert that actually splits a page requires 4 accesses plus allocating and writing the split page. Call this 5 accesses. Each insert that does not split a page requires 4 accesses.

For a static directory, each insert that splits a page reuires 3 accesses plus allocating and writing the overflow page. Call this 4 accesses. Additional inserts require 4 accesses.

As a result a static structure requires 8,000 less accesses to perform the 32,000 inserts.

Now suppose subsequently 16,000 retrieve requests for the record matching a given key are performed. In a B-tree each retrieval takes 4 accesses; for the static directory a retrieve requires 3 accesses with probability 0.5 and 4 otherwise. In this case 8,000 more accesses are saved by the static directory structure.

The point to be noted clearly is that after these 48,000 operations the static directory has saved enough accesses to allow a complete reorganization (16,000 accesses). Roughly speaking if the retrieval frequency is larger than 0.33 a static directory can satisfy all requests then be reorganized to clear all records from the overflow areas and still require less I/O activity than a B-tree. Hence, it is advantageous from a performance point of view to use a static directory in these cases.

Of course, the comparison is dependent on the assumptions made. However, it should be clear that a wide class of situations exists for which a static directory will outperform a dynamic directory.

CONCLUSIONS

In designing a data base system, one may be well aware of the problems of data base reorganization and find the dynamic directory structures very appealing. However, we suggest that the gains in ease of reorganization are not free and that cost in secondary index update, concurrency problems and height of the directory may be great.

REFERENCES

1) Bayer, R. and McCreight, E, "Organization and Maintenance of Large Ordered Indices," Proc. 1970 ACM-SIGFIDET Workshop on Data

Description, Access and Control, Houston, Texas, Nov. 1970.

2) Knuth, D., The Art of Computer Programming, Vol 3, Addison Wesley, Reading, Mass., 1973.

3) Keehn, D., and Lacy, J., "VSAM Data Set Design Parameters," IBM Systems Journal, Vol 13, No. 3, pp 186-213, 1974.

4) IBM Corp, "OS ISAM Logic," IBM, White Plains, N.Y., GY28-6618.

5) Held, G, Stonebraker, M and Wong, E., "INGRES- A Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975.

6) Stonebraker, M., "The Choice of Partial Inversions and Combined Indices," International Journal of Computer and Information Science, June 1974.

7) Held, G. and Stonebraker, M., "Storage Structures and Access Methods in the Relational Data Base Management System, INGRES," Proc. 1975 ACM-PACIFIC, San Francisco, Ca., April 1975.

8) Ritchie, D. and Thompson, K., "The UNIX Time Sharing System," CACM, Vol 17, No. 7, pp 365-375, July 1974.

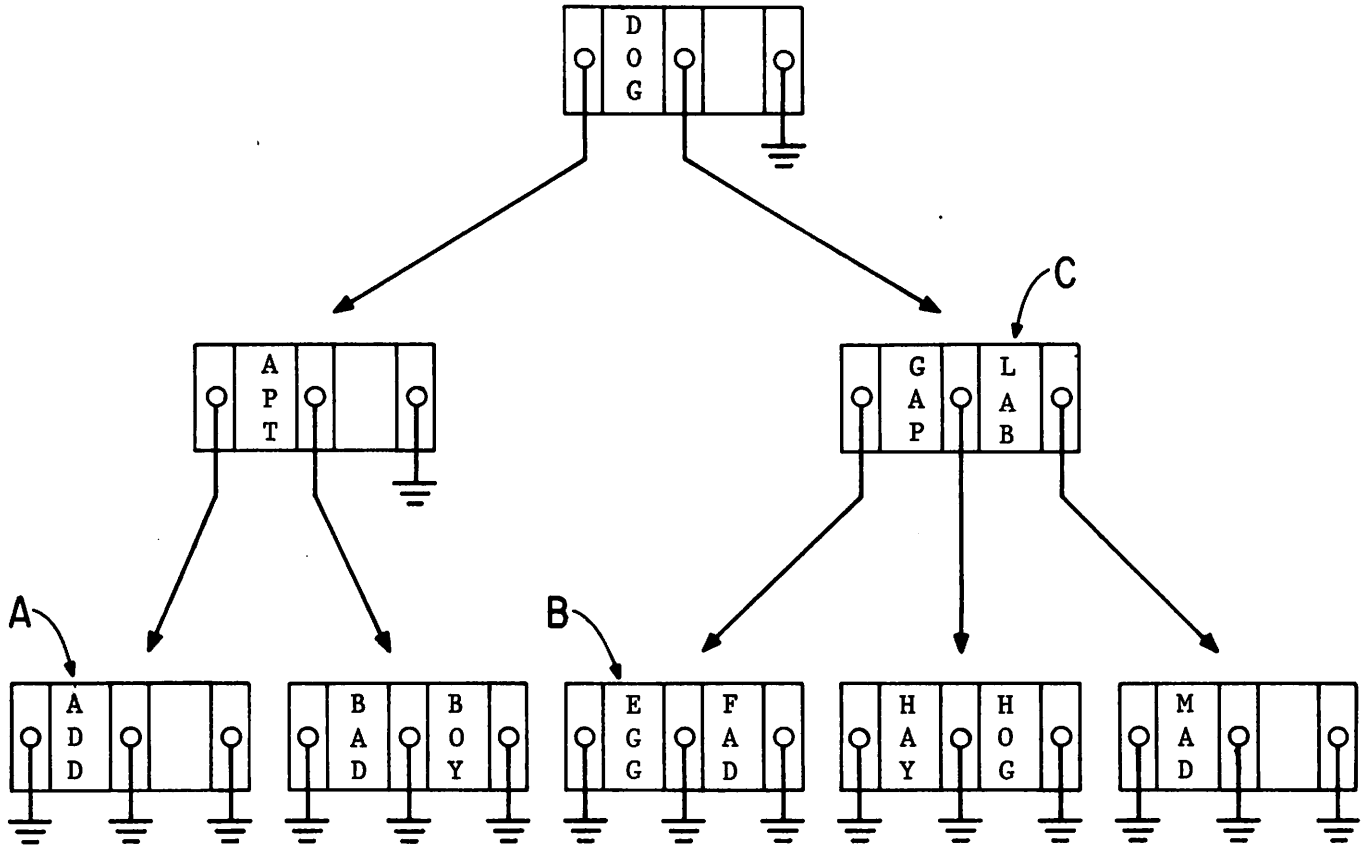

Fig. 1. A B-tree.

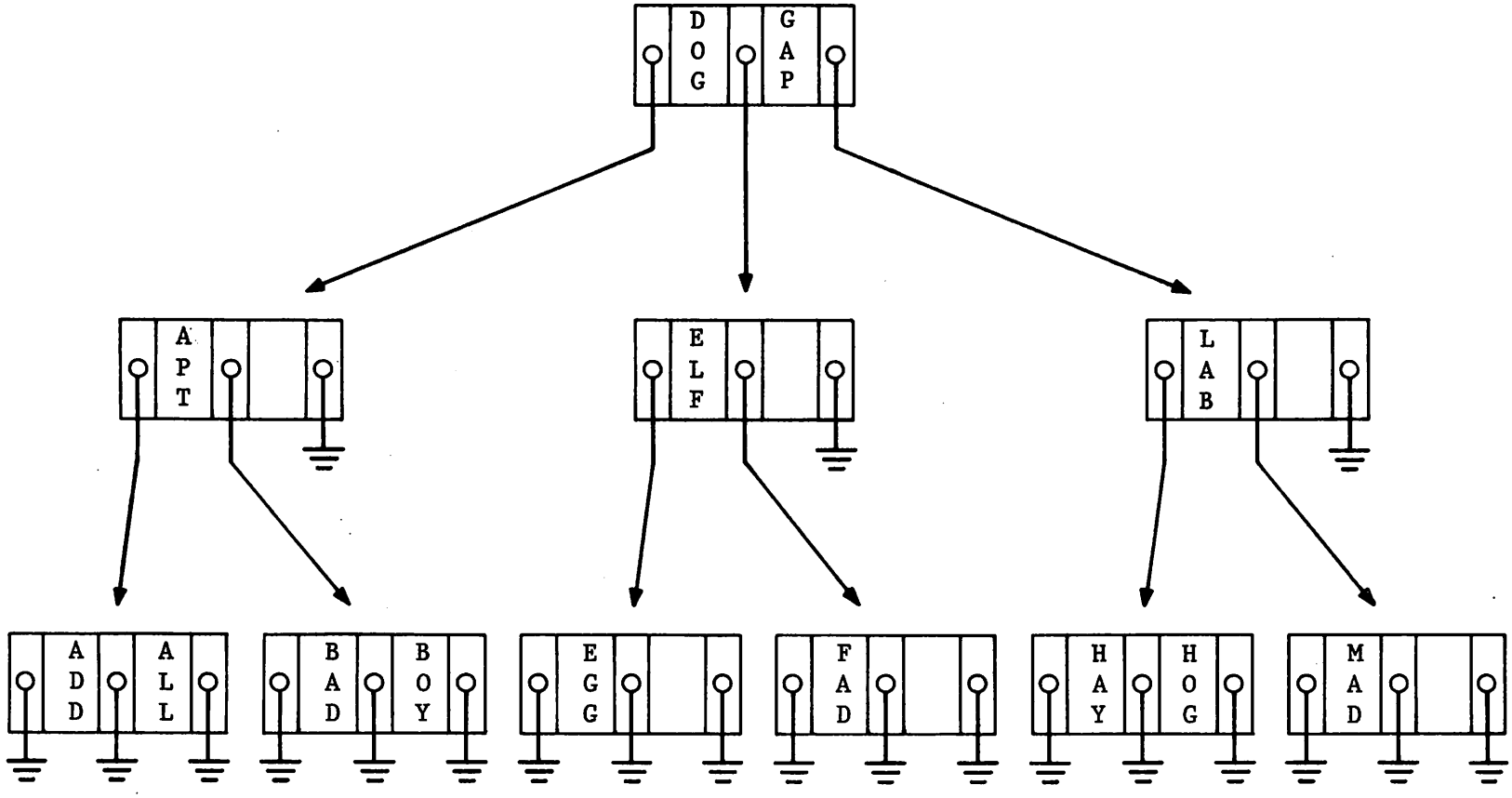

Fig. 2. The Updated B-tree.

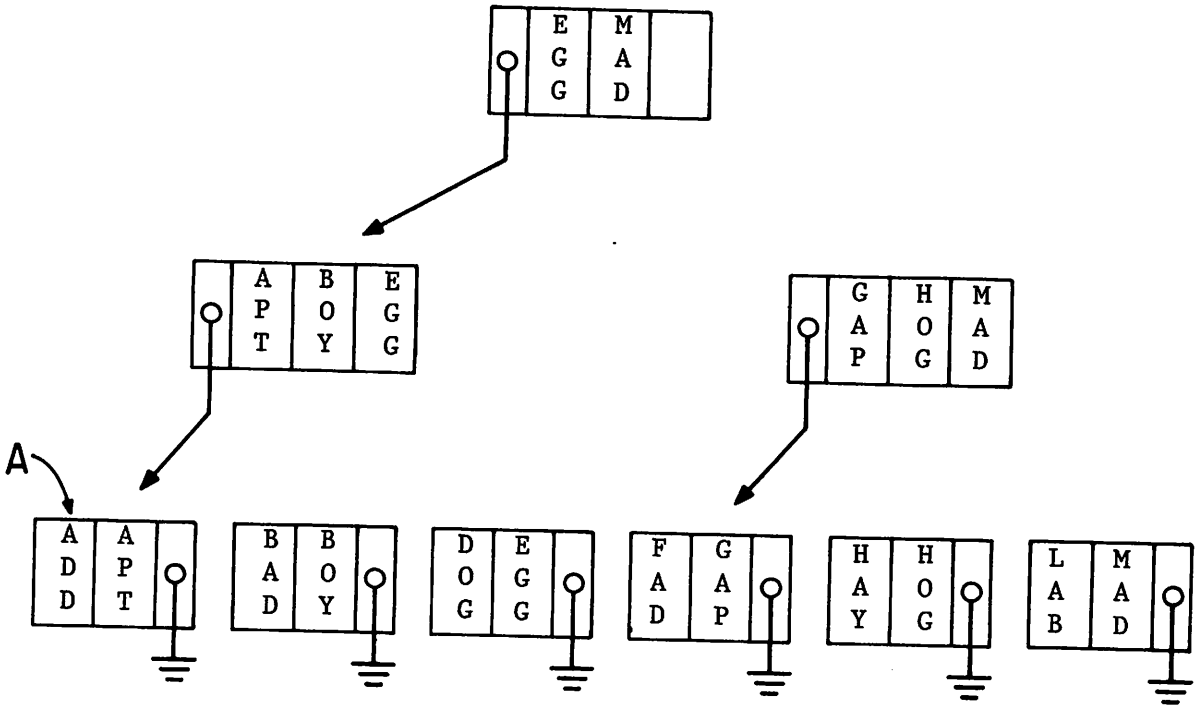

Fig. 3. A Static Structure.

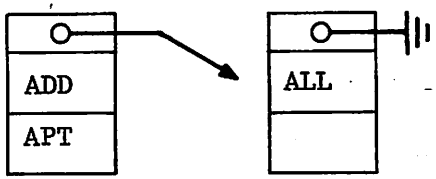

Fig. 4. A Portion of the Updated Static Structure.

B	→ LAB
D	→ BAD
D	→ ADD
D	→ MAD
D	→ FAD
G	→ HOG
G	→ EGG
G	→ DOG
P	→ GAP
T	→ APT
Y	→ BOY
Y	→ HAY

Fig. 5. An Inversion on the Last Letter of a Key.